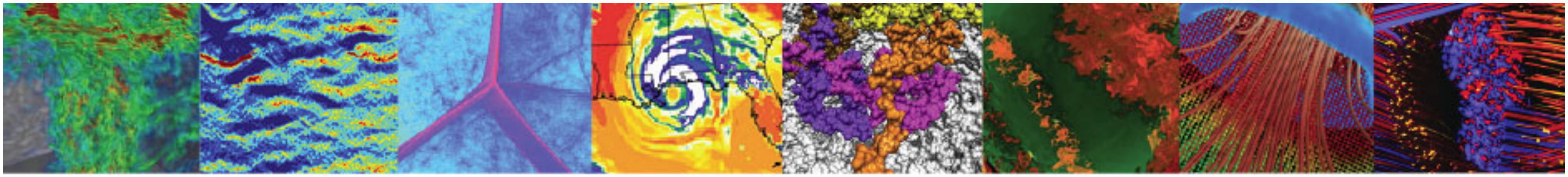


Argonne Training Program on
EXTREME-SCALE COMPUTING

Argonne NATIONAL LABORATORY

July 28 – August 9, 2013

Adaptive Linear Solvers and Eigensolvers

Jack Dongarra

University of Tennessee
Oak Ridge National Laboratory
University of Manchester

Dense Linear Algebra

- **Common Operations**

$$Ax = b; \quad \min_x \| Ax - b \|; \quad Ax = \lambda x$$

- **A major source of large dense linear systems is problems involving the solution of boundary integral equations.**
  - **The price one pays for replacing three dimensions with two is that what started as a sparse problem in *$O(n^3)$* variables is replaced by a dense problem in *$O(n^2)$*.**
- **Dense systems of linear equations are found in numerous other applications, including:**
  - **airplane wing design;**
  - **radar cross-section studies;**
  - **flow around ships and other off-shore constructions;**
  - **diffusion of solid bodies in a liquid;**
  - **noise reduction; and**
  - **diffusion of light through small particles.**

Existing Math Software - Dense LA

DIRECT SOLVERS	License	Support	Type		Language			Mode		
			Real	Complex	F77	C	C++	Shared	GPU	Dist
Eigen	Mozilla	yes	X	X			X	X		
Elemental	BSD	yes	X	X			X			M
FLAME	LGPL	yes	X	X	X	X		X		
FLENS	BSD	yes	X	X			X	X		
LAPACK	BSD	yes	X	X	X	X		X		
LAPACK95	BSD	yes	X	X	F95			X		
MAGMA	BSD	yes	X	X	X	X		X	C/O/X	
NAPACK	BSD	yes	X		X			X		
PLAPACK	?	no	X	X	X	X				M
PLASMA	BSD	yes	X	X	X	X		X		
PRISM	?	no	X		X			X		M
rejtrix	by-nc-sa	yes	X				X	X		
ScaLAPACK	BSD	yes	X	X	X	X				M/P
Trilinos/Pliris	BSD	yes	X	X		X	X			M
ViennaCL	MIT	yes	X				X	X	C/O/X	

http://www.netlib.org/utk/people/JackDongarra/la-sw.html

- **LINPACK, EISPACK, LAPACK, ScaLAPACK**
  - **PLASMA, MAGMA**

June 2013: The TOP10

Rank	Site	Computer	Country	Cores	Rmax [Pflops]	% of Peak	Power [MW]	MFlops /Watt
1	National University of Defense Technology	Tianhe-2 NUDT, Xeon 12C 2.2GHz + IntelXeon Phi (57c) + Custom	China	3,120,000	33.9	70	17.8	1905
2	DOE / OS Oak Ridge Nat Lab	Titan, Cray XK7 (16C) + Nvidia Kepler GPU (14c) + Custom	USA	560,640	17.6	66	8.3	2120
3	DOE / NNSA L Livermore Nat Lab	Sequoia, BlueGene/Q (16c) + custom	USA	1,572,864	16.3	81	7.9	2063
4	RIKEN Advanced Inst for Comp Sci	K computer Fujitsu SPARC64 VIIIfx (8c) + Custom	Japan	705,024	10.5	93	12.7	827
5	DOE / OS Argonne Nat Lab	Mira, BlueGene/Q (16c) + Custom	USA	786,432	8.16	81	3.95	2066
6	Texas Advanced Computing Center	Stampede, Dell Intel (8c) + Intel Xeon Phi (61c) + IB	USA	204,900	2.66	67	3.3	806
7	Forschungszentrum Juelich (FZJ)	JuQUEEN, BlueGene/Q, Power BQC 16C 1.6GHz+Custom	Germany	458,752	5.01	85	2.30	2178
8	DOE / NNSA L Livermore Nat Lab	Vulcan, BlueGene/Q, Power BQC 16C 1.6GHz+Custom	USA	393,216	4.29	85	1.97	2177
9	Leibniz Rechenzentrum	SuperMUC, Intel (8c) + IB	Germany	147,456	2.90	90*	3.42	848
10	Nat. SuperComputer Center in Tianjin	Tianhe-1A, NUDT Intel (6c) + Nvidia Fermi GPU (14c) + Custom	China	186,368	2.57	55	4.04	636
500	US Navy DSRC	Cray XT5	USA	12,720	.096	79		

Potential System Architecture with a cap of $200M and 20MW

Systems	2013 Tianhe-2	2022	Difference Today & 2022
System peak	**55 Pflop/s**	**1 Eflop/s**	**~20x**
Power	**18 MW** **(3 Gflops/W)**	**~20 MW** **(50 Gflops/W)**	**O(1)** **~15x**
System memory	1.4 PB (1.024 PB CPU + .384 PB CoP)	32 - 64 PB	~50x
Node performance	3.43 TF/s (.4 CPU +3 CoP)	1.2 or 15TF/s	O(1)
Node concurrency	24 cores CPU + 171 cores CoP	O(1k) or 10k	~5x - ~50x
Node Interconnect BW	6.36 GB/s	200-400GB/s	~40x
System size (nodes)	16,000	O(100,000) or O(1M)	~6x - ~60x
Total concurrency	3.12 M 12.48M threads (4/core)	O(billion)	~100x
MTTF	?? unknown	O(<1 day)	O(?)

Factors that Necessitate Redesign

- Steepness of the ascent from terascale to petascale to exascale
- Extreme parallelism and hybrid design
  - Preparing for million/billion way parallelism
- Tightening memory/bandwidth bottleneck
  - Limits on power/clock speed implication on multicore
  - Reducing communication will become much more intense
  - Memory per core changes, byte-to-flop ratio will change
- Necessary Fault Tolerance
  - MTTF will drop
  - Checkpoint/restart has limitations

Key Challenges at Exascale

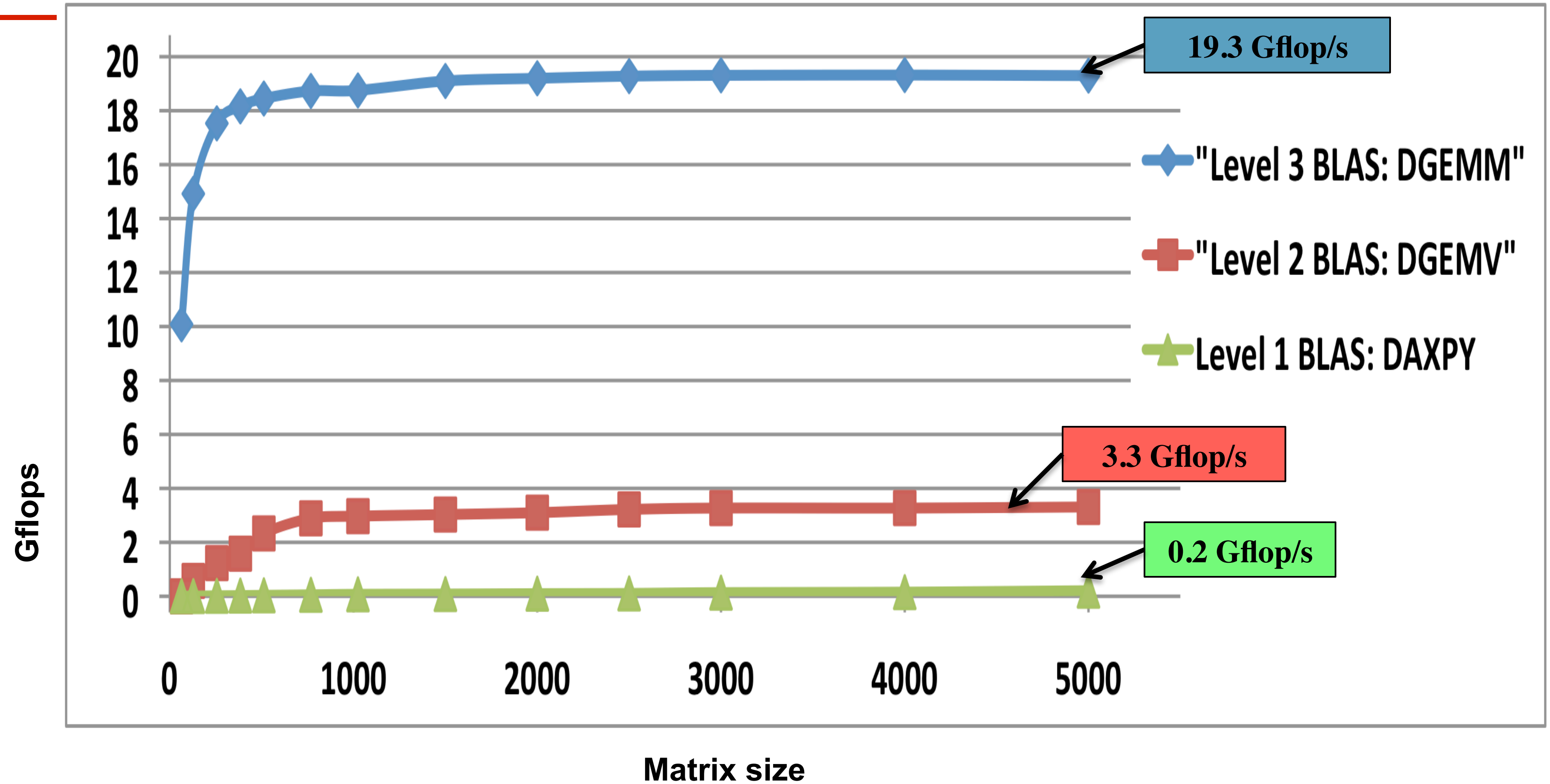
- Levels of parallelism
  - O(100M and beyond)
- Hybrid architectures
  - Node composed of multiple multicore sockets + accelerators
- Bandwidth vs Arithmetic rate
  - Most approaches assume flops expensive
- Storage Capacity
  - Issue of weak scalability in future systems
- Fault occurrence; shared responsibility
  - Process failure recovery
- Power Management
  - API for fine grain management
- Language constraints
  - Fortran, C & MPI, Open-MP
- Autotuning
  - Systems complex and changing
- Bulk Sync Processing
  - Break fork join parallelism
- Lack of reproducibility; unnecessarily expensive (most of the time)
  - Can't guarantee bitwise results
- Need for effective scheduling of tasks

Critical Issues at Peta & Exascale for Algorithm and Software Design

- Synchronization-reducing algorithms
  - Break Fork-Join model
- Communication-reducing algorithms
  - Use methods which have lower bound on communication
  - Cache aware
- Mixed precision methods
  - 2x speed of ops and 2x speed for data movement
- Autotuning
  - Today's machines are too complicated, build "smarts" into software to adapt to the hardware
- Fault resilient algorithms
  - Implement algorithms that can recover from failures/bit flips
- Reproducibility of results
  - Today we can't guarantee this. We understand the issues, but some of our "colleagues" have a hard time with this.

Level 1, 2 and 3 BLAS

1 core Intel Xeon E5-2670 (Sandy Bridge); 2.6 GHz; Peak = 20.8 Gflop/s



1 core Intel Xeon E5-2670 (Sandy Bridge), 2.6 GHz.
24 MB shared L3 cache, and each core has a private 256 KB L2 and 64 KB L1.
The theoretical peak per core DP is 8 flop/cycle * 2.6 GHz = 20.8 Gflop/s per core.
Compiled with gcc 4.4.6 and using MKL_composer_xe_2013.3.163

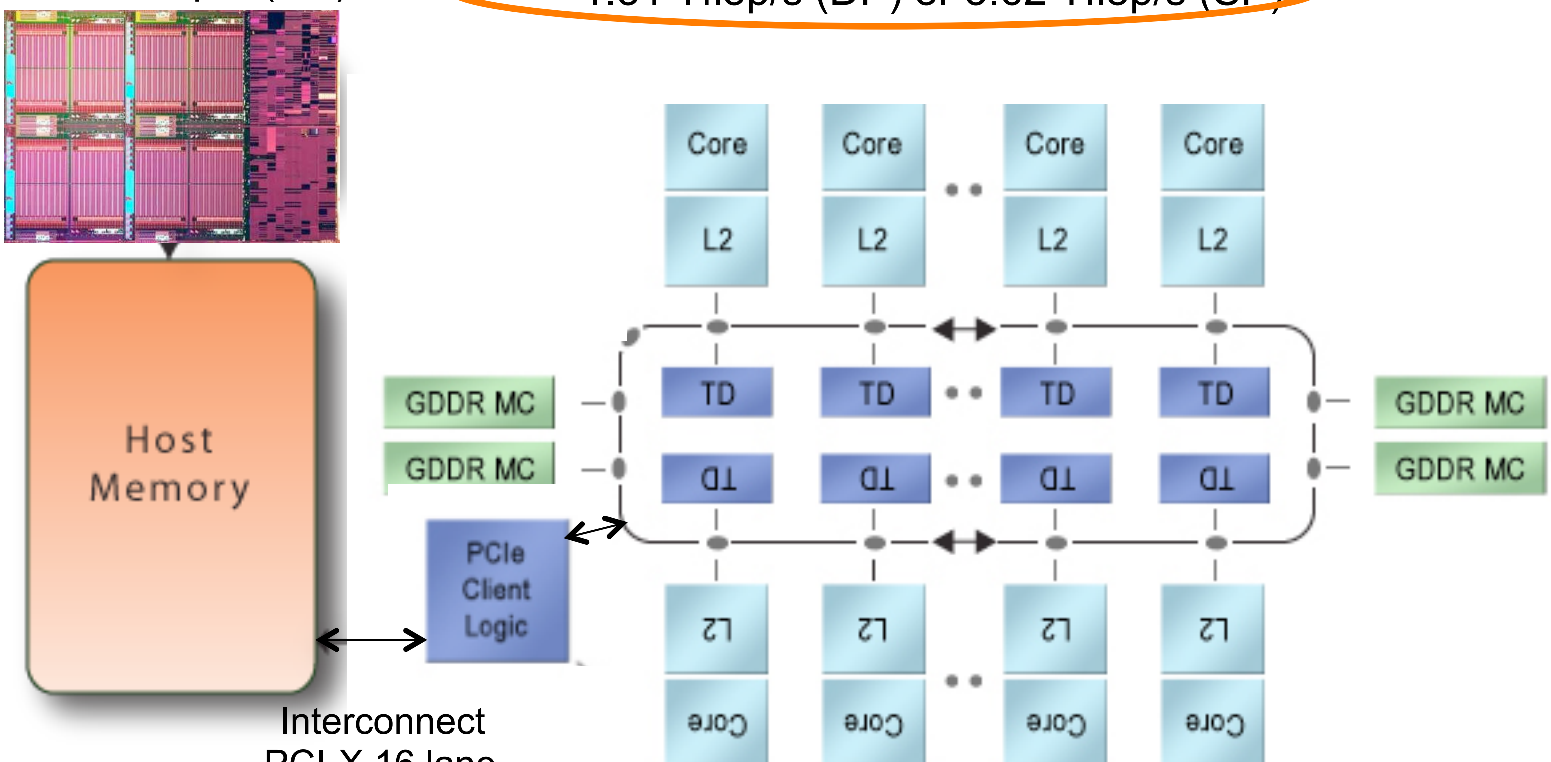
Commodity plus Accelerator Today

Commodity

Intel Sandy Bridge
8 cores
2.6 GHz
8*2.6*8 ops/cycle
166.4 Gflop/s (DP)

Accelerator/Co-Processor

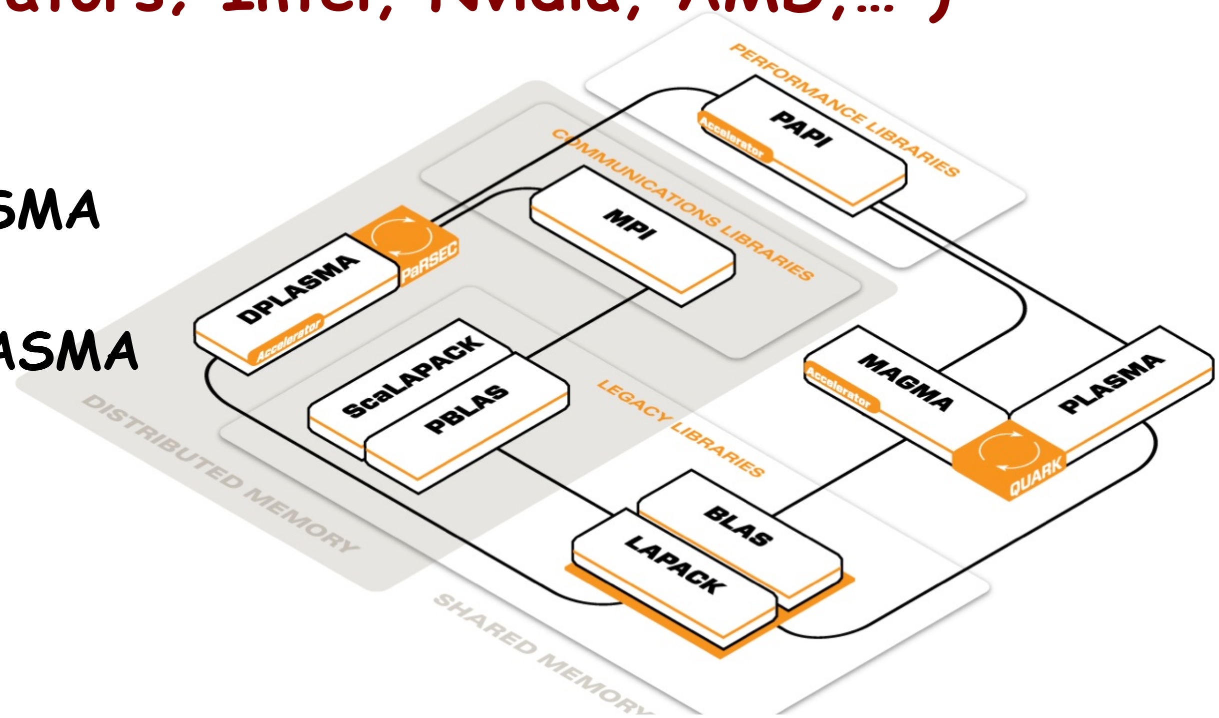
Intel Xeon Phi
244 “cores” (4 used by OS)
61 (60) FPU = 61 (60) cores
1.091 GHz
60*1.092*8*2 ops/cycle
1.31 Tflop/s (DP) or 3.62 Tflop/s (SP)



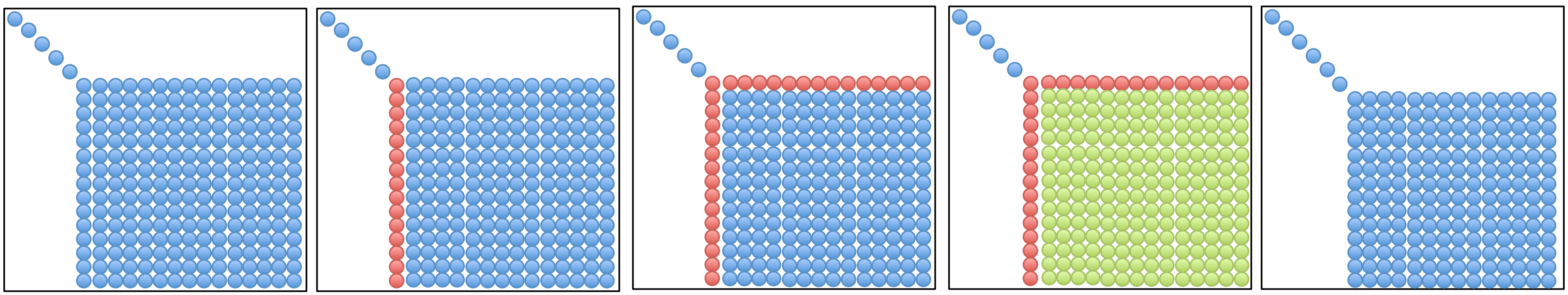
Interconnect
PCI-X 16 lane
64 Gb/s (8 GB/s)
1 GW/s

Dense Linear Algebra

- **Numerical Linear Algebra Algorithms and Software**
  - EISPACK, LINPACK, BLAS, LAPACK, ScaLAPACK, PBLAS, ATLAS
  - PLASMA: Manycore; DPLASMA: Distributed)
  - MAGMA (Accelerators; Intel, Nvidia, AMD,…)
  - QUARK
    - Runtime for PLASMA
  - PaRSEC
    - Runtime for DPLASMA



The Standard LU Factorization LINPACK
1970's HPC of the Day: Vector Architecture



Factor column with Level 1 BLAS

Divide by Pivot row

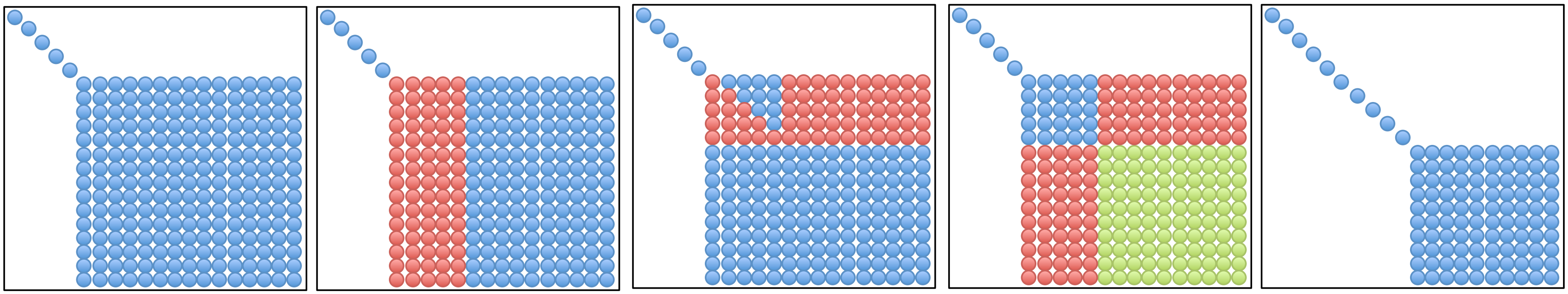
Schur complement update (Rank 1 update)

Next Step

Main points

- Factorization column (zero) mostly sequential due to memory bottleneck
- Level 1 BLAS
- Divide pivot row has little parallelism
- Rank -1 Schur complement update is the only easy parallelize task
- Partial pivoting complicates things even further
- Bulk synchronous parallelism (fork-join)
    - Load imbalance
    - Non-trivial Amdahl fraction in the panel
    - Potential workaround (look-ahead) has complicated implementation

The Standard LU Factorization LAPACK 1980's HPC of the Day: Cache Based SMP



Factor panel with Level 1,2 BLAS

Triangular update

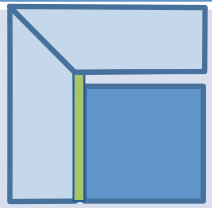
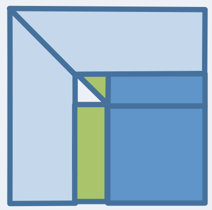
Schur complement update

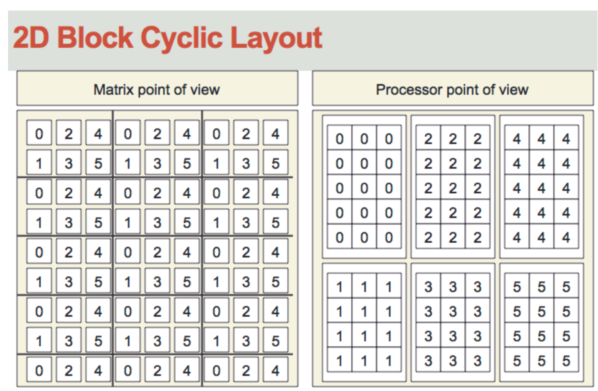
Next Step

Main points

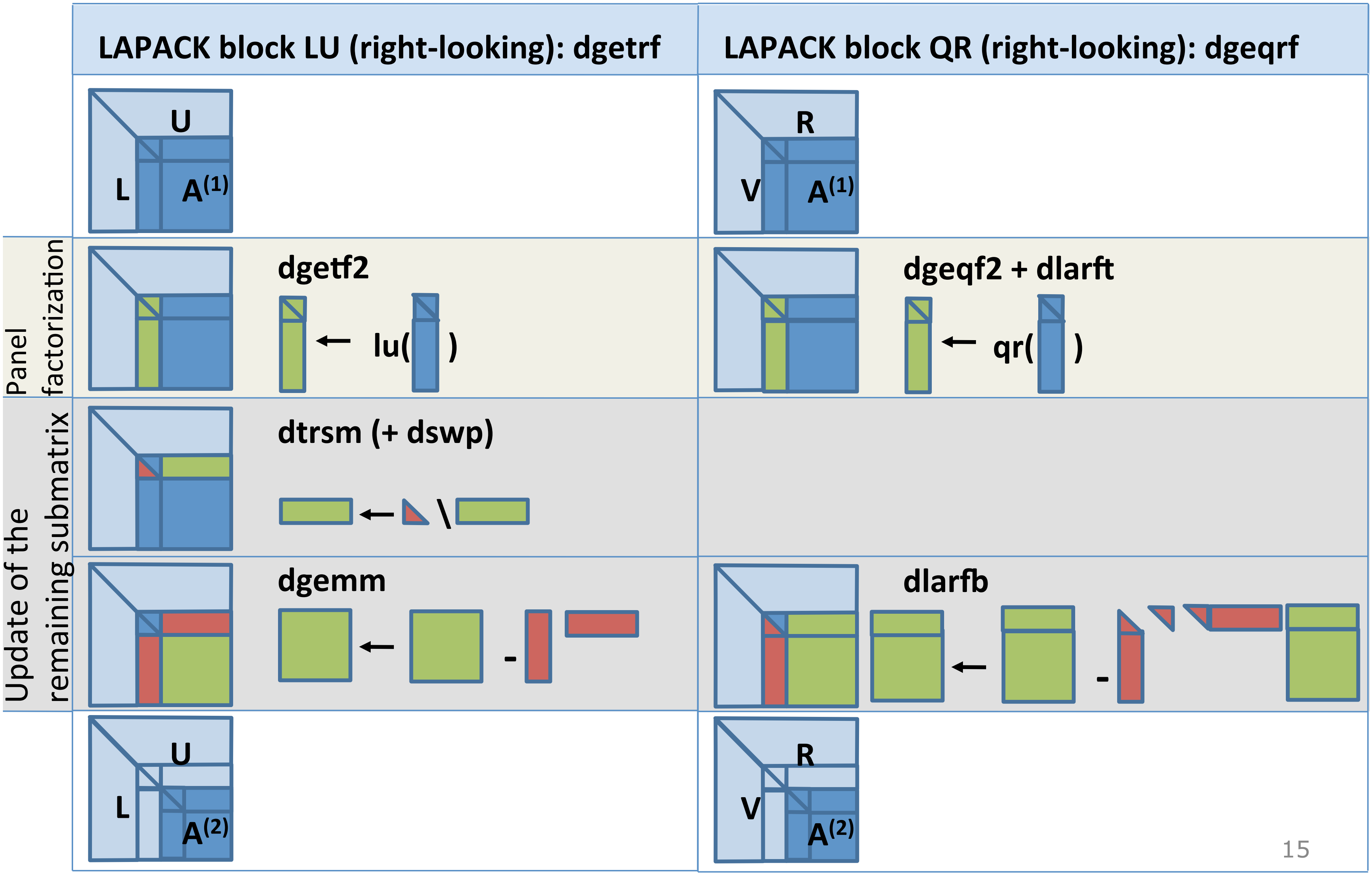
- Panel factorization mostly sequential due to memory bottleneck
- Triangular solve has little parallelism
- Schur complement update is the only easy parallelize task
- Partial pivoting complicates things even further
- Bulk synchronous parallelism (fork-join)
    - Load imbalance
    - Non-trivial Amdahl fraction in the panel
    - Potential workaround (look-ahead) has complicated implementation

A New Generation of DLA Software

Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (80's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations
ScaLAPACK (90's) (Distributed Memory)		Rely on - PBLAS Mess Passing



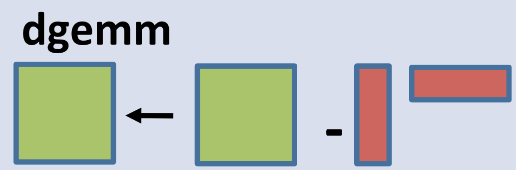
Blocked LU and QR algorithms (LAPACK)

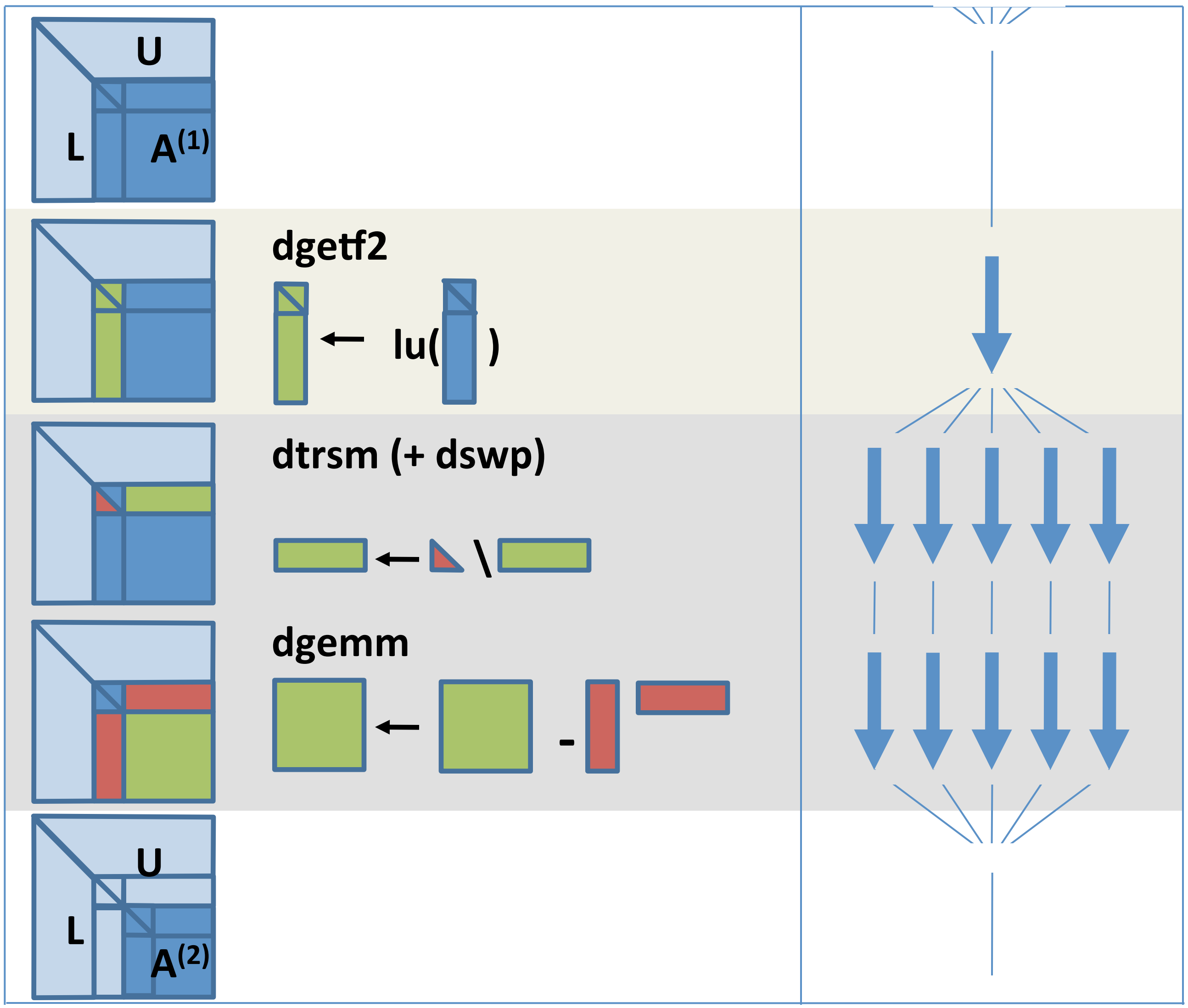


Parallelization of LU and QR.

Parallelize the update:

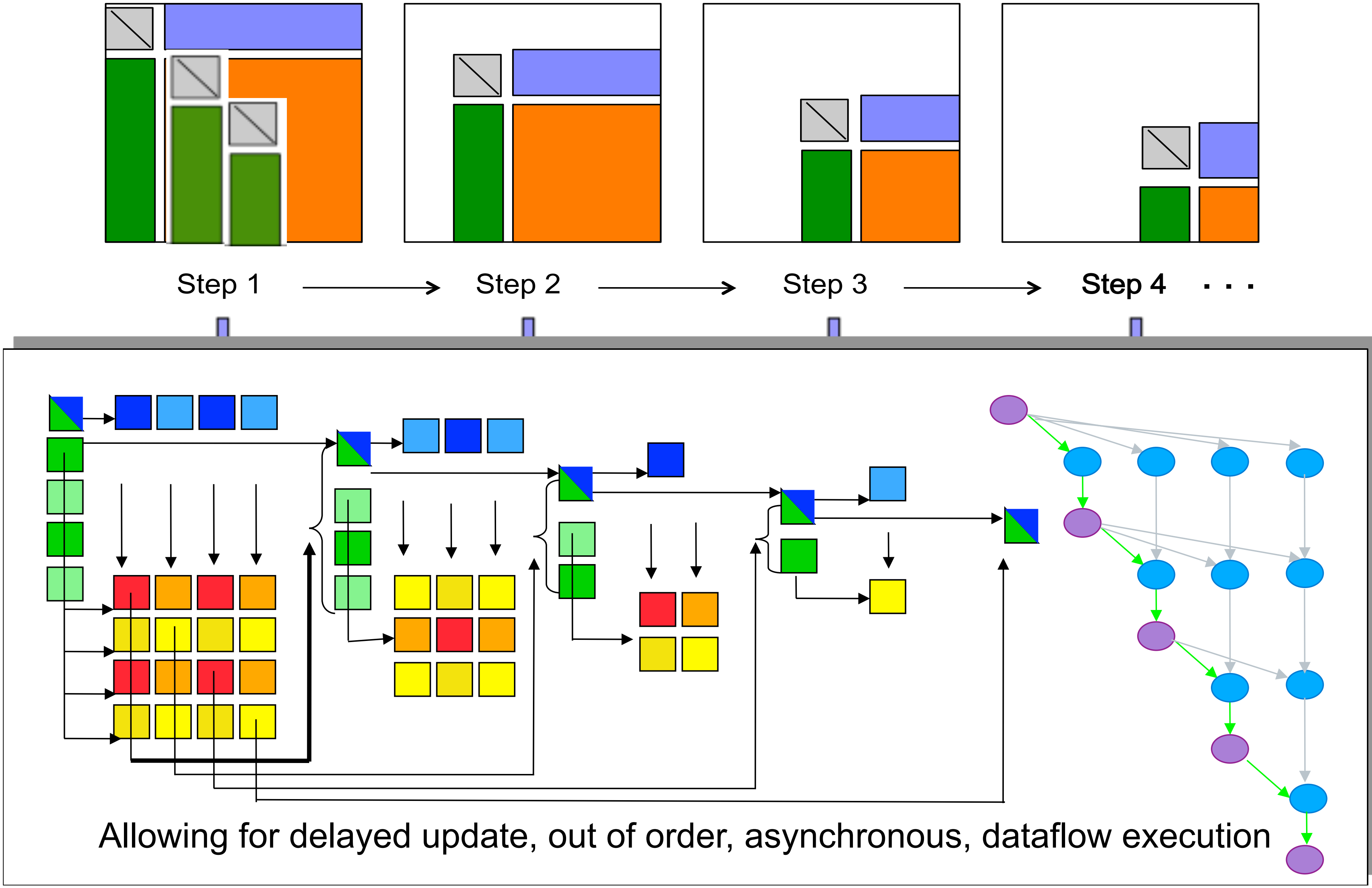
- Easy and done in any reasonable software.
- This is the $2/3n^3$ term in the FLOPs count.
- Can be done efficiently with LAPACK+multithreaded BLAS





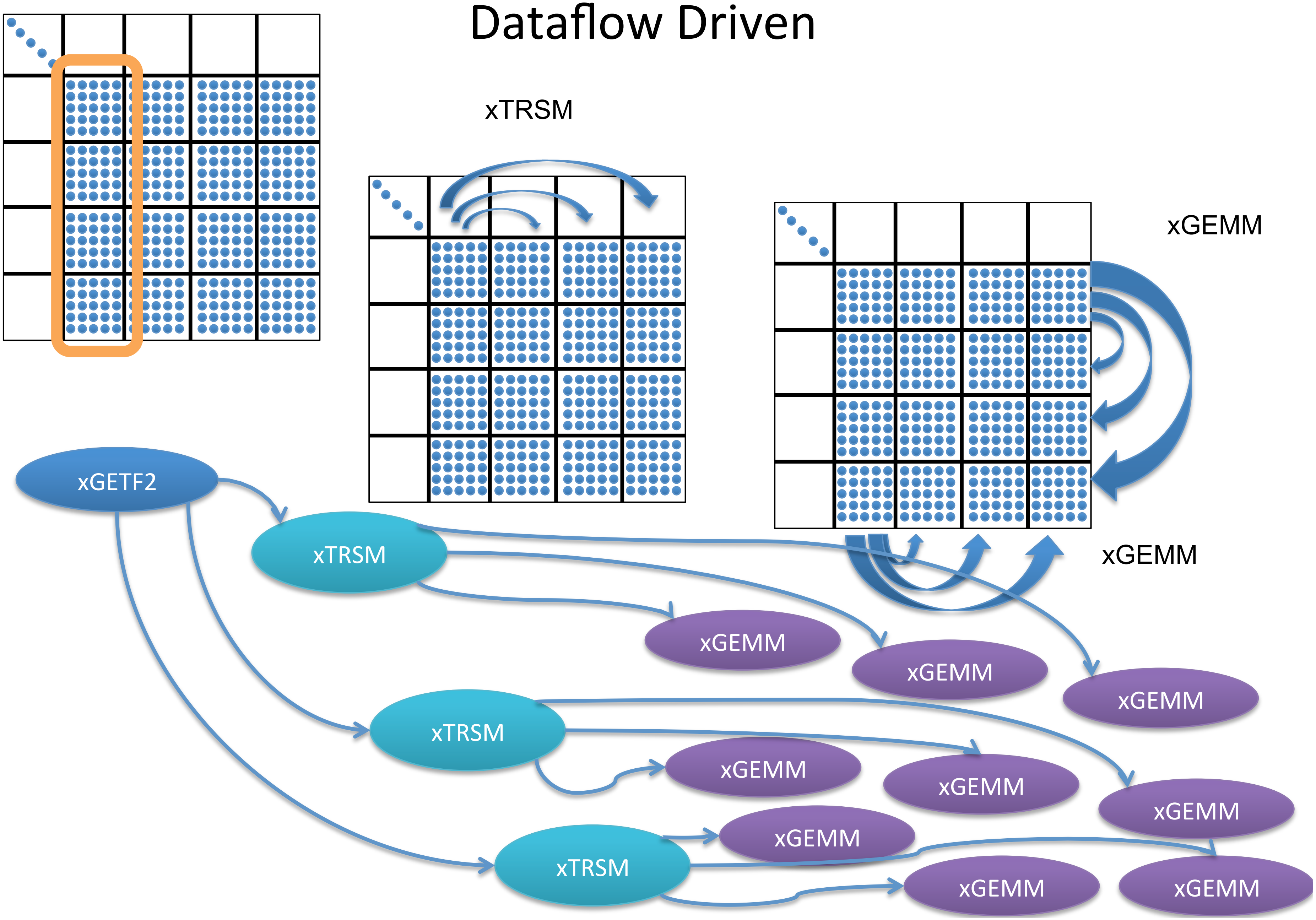
Fork - Join parallelism
Bulk Sync Processing

Synchronization (in LAPACK LU)

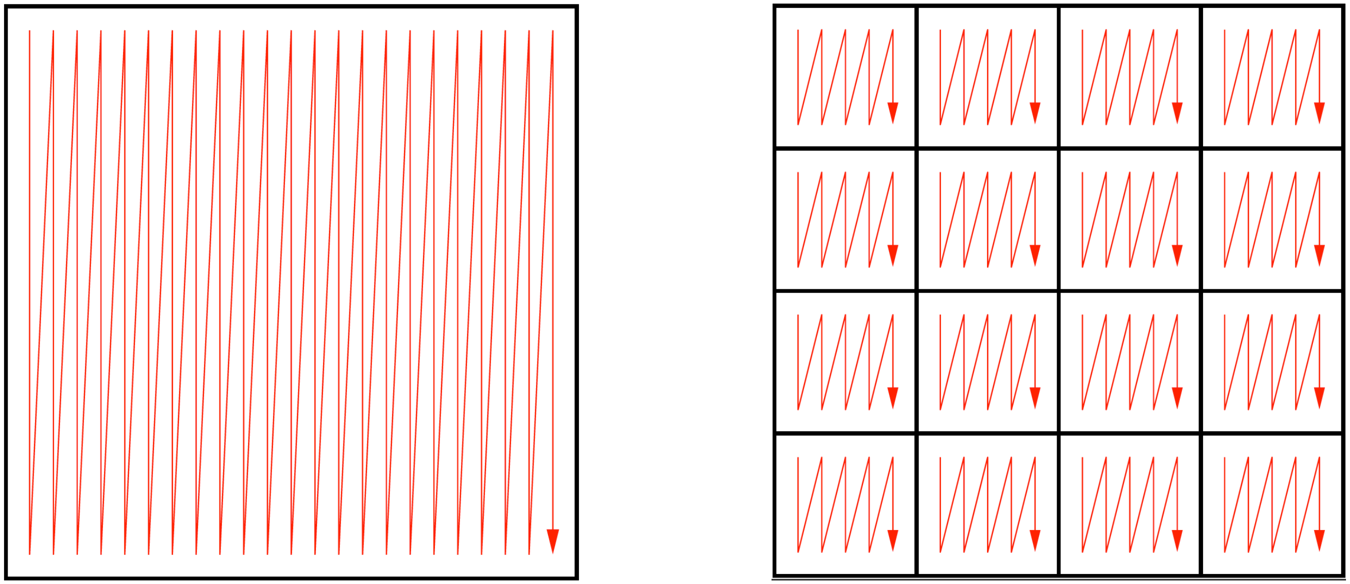


PLASMA LU Factorization

Dataflow Driven



Data Layout is Critical



- Tile data layout where each data tile is contiguous in memory
- Decomposed into several fine-grained tasks, which better fit the memory of the small core caches

PLASMA LU: Tile Algorithm and Nested Parallelism

- Operates on one, two, or three matrix tiles at a time using a single core
  - This is called a kernel; executed independently of other kernels
  - Mostly Level 3 BLAS are used
- Data flows between kernels as prescribed by the programmer
- Coordination is done transparently via runtime scheduler (QUARK)
  - Parallelism level adjusted at runtime
  - Look-ahead adjusted at runtime
- Uses single-threaded BLAS with all the optimization benefits
- Panel is done on multiple cores
  - Recursive formulation of LU for better BLAS use
  - Level 1 BLAS are faster because they work on combined cache size

QUARK

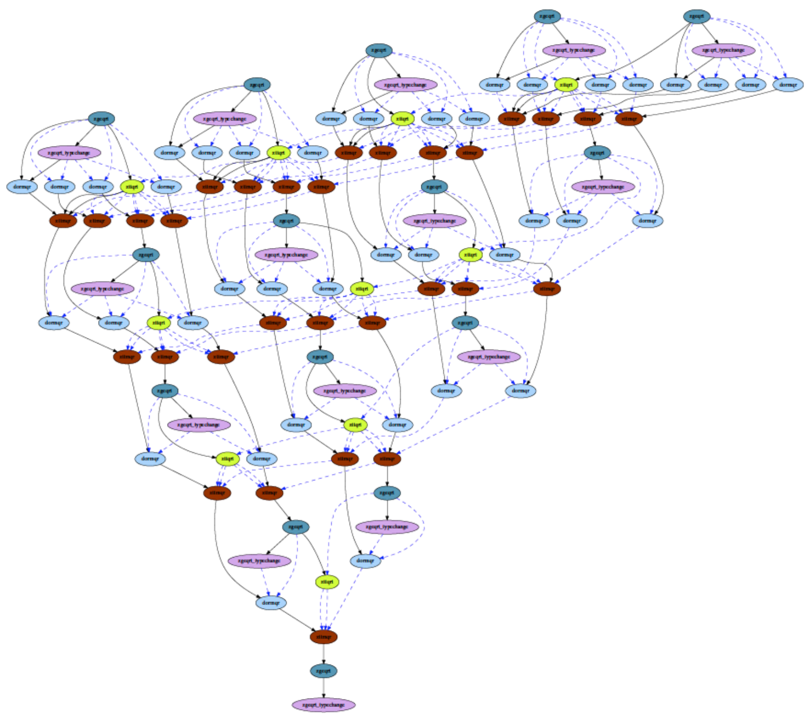
Shared Memory Superscalar Scheduling

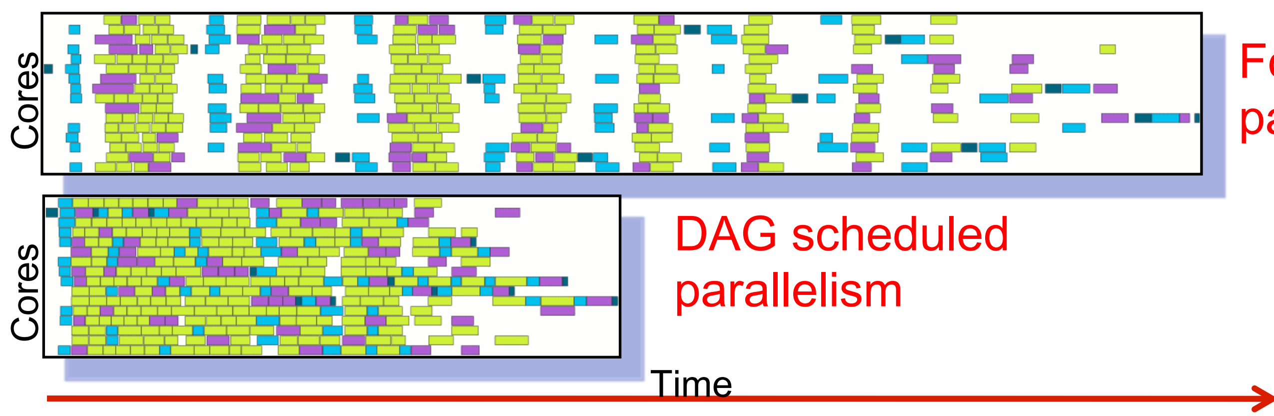
definition – pseudocode

```
FOR k = 0..TILES-1
  A[k][k] ← DPOTRF(A[k][k])
  FOR m = k+1..TILES-1
    A[m][k] ← DTRSM(A[k][k], A[m][k])
  FOR m = k+1..TILES-1
    A[m][m] ← DSYRK(A[m][k], A[m][m])
    FOR n = k+1..m-1
      A[m][n] ← DGEMM(A[m][k], A[n][k], A[m][n])
```

Parallel Linear Algebra s/w for Multicore/Hybrid Architectures

- **Objectives**
  - High utilization of each core
  - Scaling to large number of cores
  - Synchronization reducing algorithms
- **Methodology**
  - Dynamic DAG scheduling (QUARK)
  - Explicit parallelism
  - Implicit communication
  - Fine granularity / block data layout
- **Arbitrary DAG with dynamic scheduling**



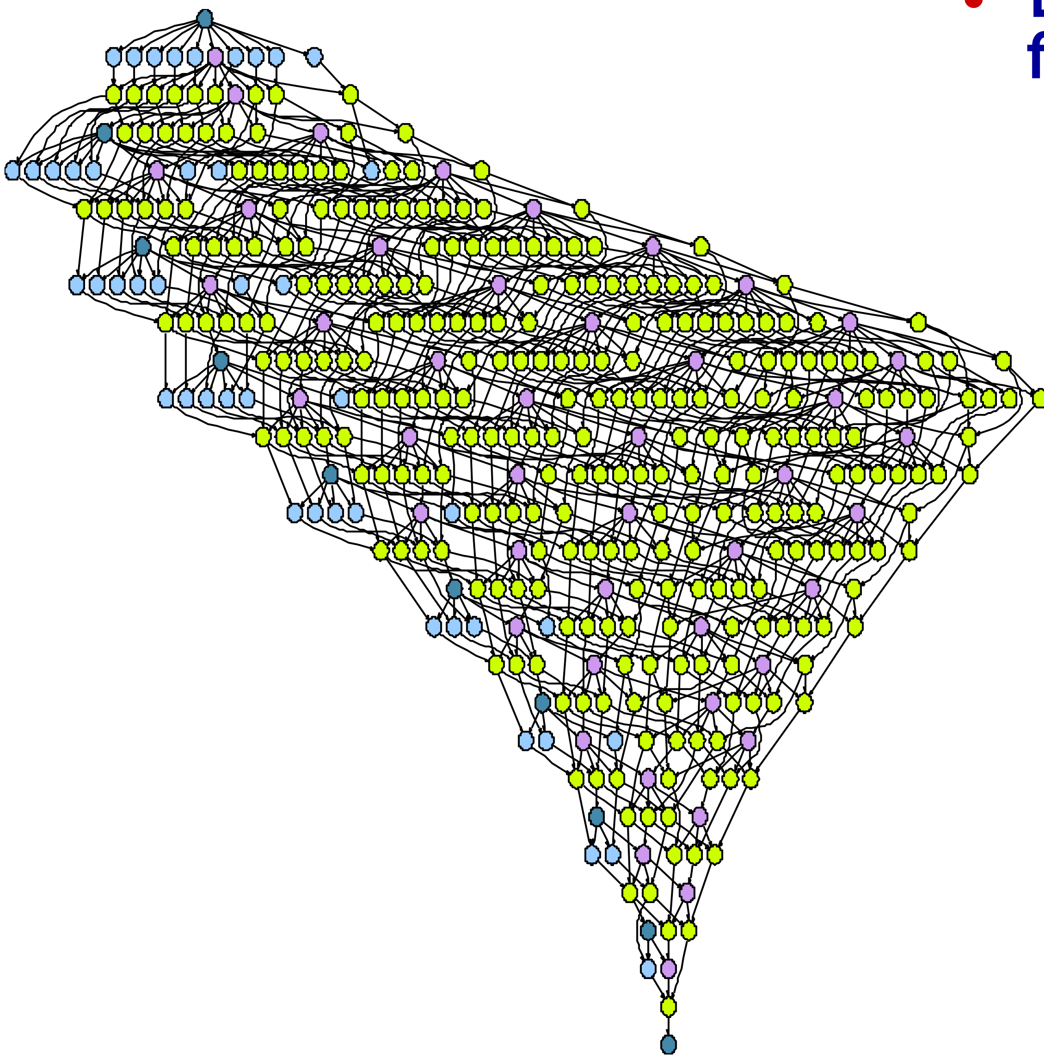


Fork-join parallelism

DAG scheduled parallelism

PLASMA Local Scheduling

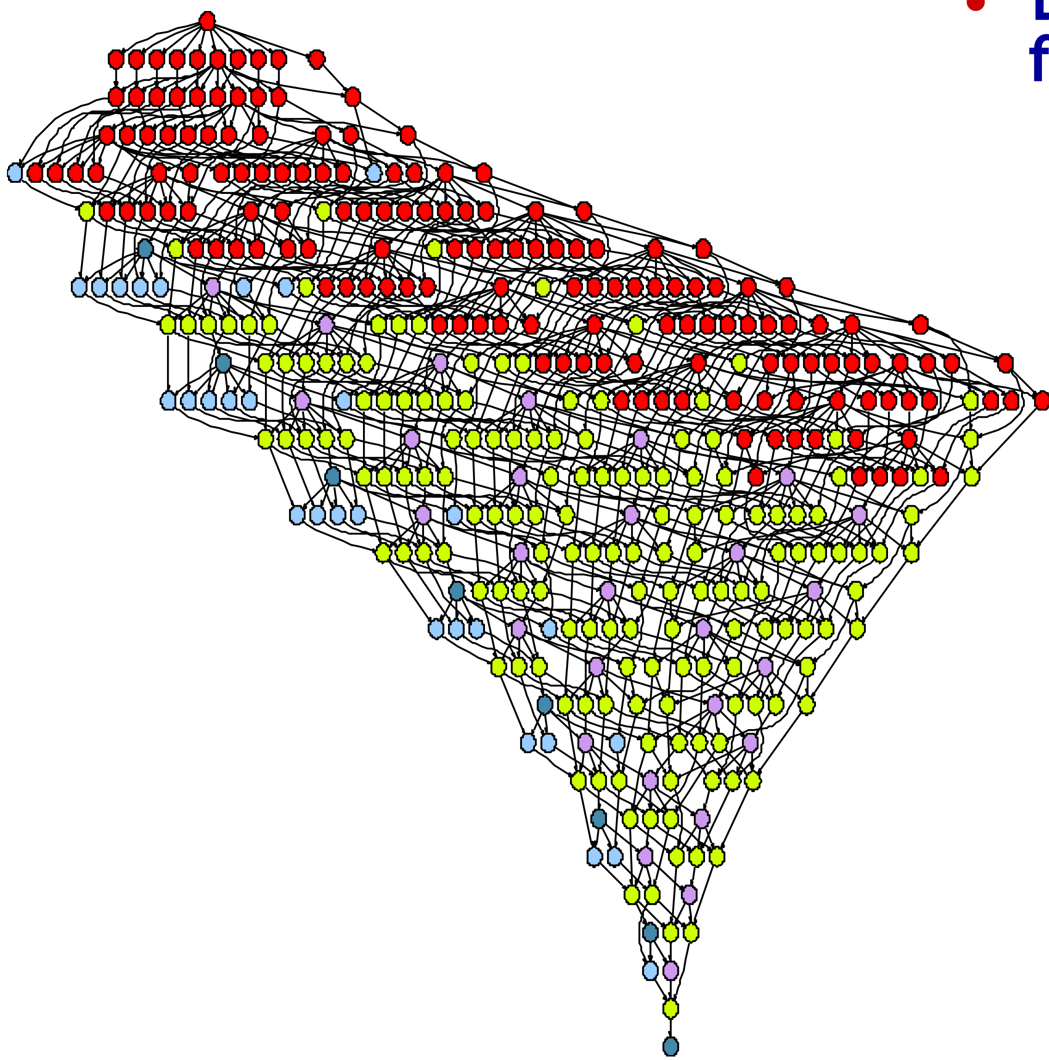
Dynamic Scheduling: Sliding Window



- DAGs get very big, very fast
  - So windows of active tasks are used; this means no global critical path
  - Matrix of NBxNB tiles; NB^3 operation
    - NB=100 gives 1 million tasks

PLASMA Local Scheduling

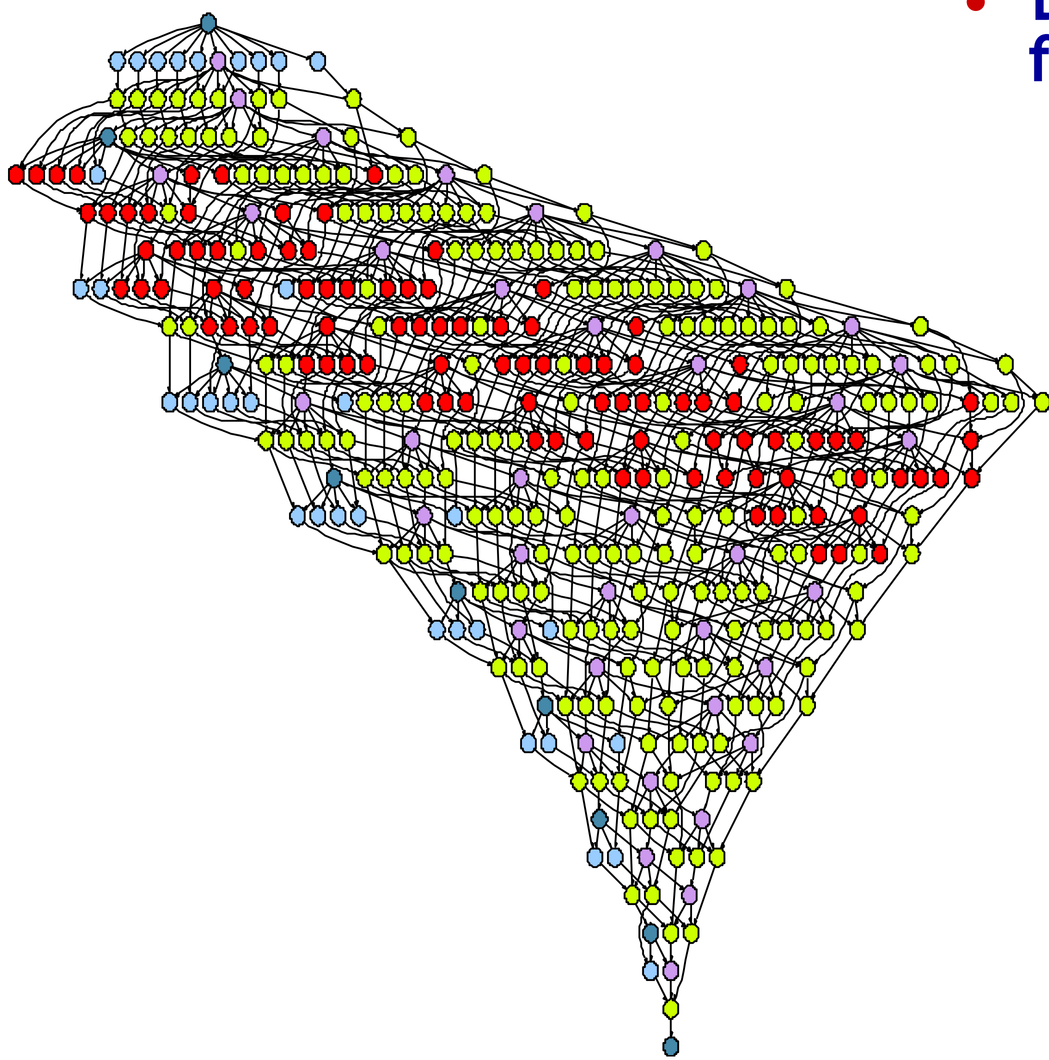
Dynamic Scheduling: Sliding Window



- **DAGs get very big, very fast**
  - **So windows of active tasks are used; this means no global critical path**
  - **Matrix of NBxNB tiles; NB^3 operation**
    - **NB=100 gives 1 million tasks**

PLASMA Local Scheduling

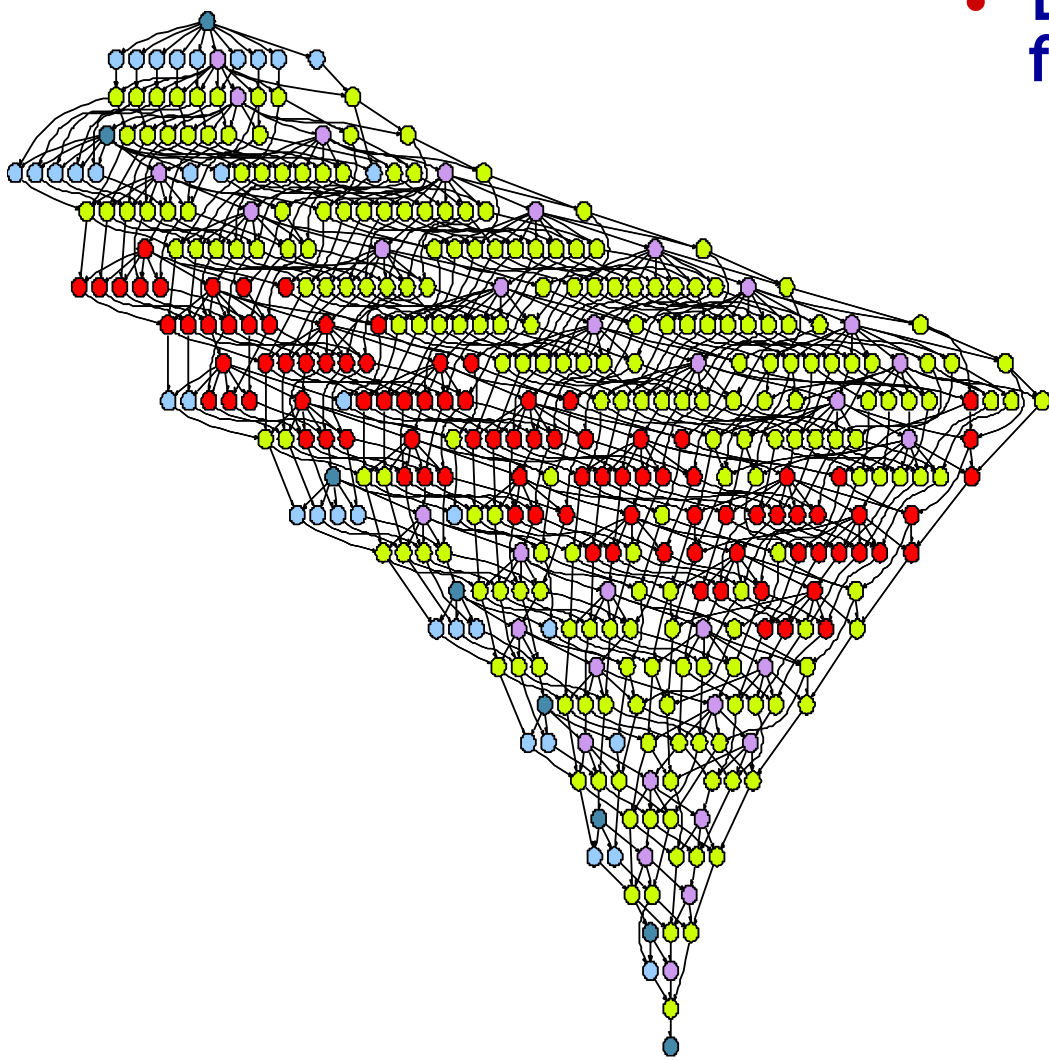
Dynamic Scheduling: Sliding Window



- **DAGs get very big, very fast**
  - **So windows of active tasks are used; this means no global critical path**
  - **Matrix of NBxNB tiles; NB^3 operation**
    - **NB=100 gives 1 million tasks**

PLASMA Local Scheduling

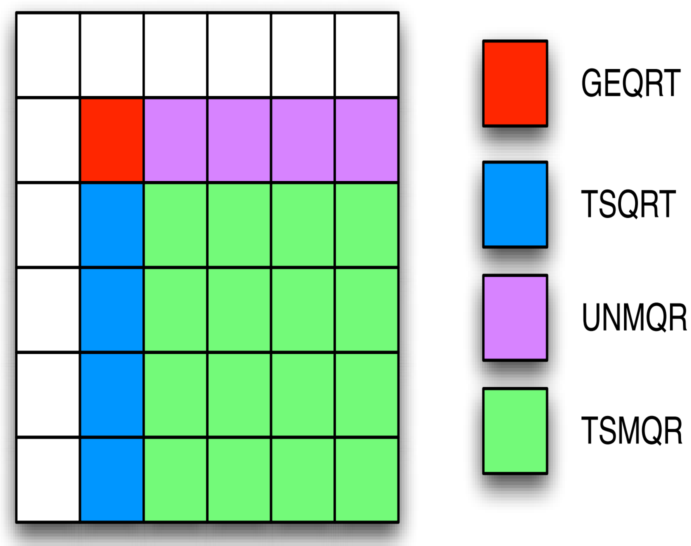
Dynamic Scheduling: Sliding Window



- DAGs get very big, very fast
  - So windows of active tasks are used; this means no global critical path
  - Matrix of NBxNB tiles; NB^3 operation
    - NB=100 gives 1 million tasks

Example: QR Factorization

```
FOR k = 0 .. SIZE - 1

    A[k][k], T[k][k]  <-  GEQRT( A[k][k] )

    FOR m = k+1 .. SIZE - 1

        A[k][k]|Up, A[m][k], T[m][k]  <-
            TSQRT( A[k][k]|Up, A[m][k], T[m][k] )

    FOR n = k+1 .. SIZE - 1

        A[k][n] <- UNMQR( A[k][k]|Low, T[k][k], A[k][n] )

        FOR m = k+1 .. SIZE - 1

            A[k][n], A[m][n] <-
                TSMQR( A[m][k], T[m][k], A[k][n], A[m][n] )
```



Input Format – Quark (PLASMA)

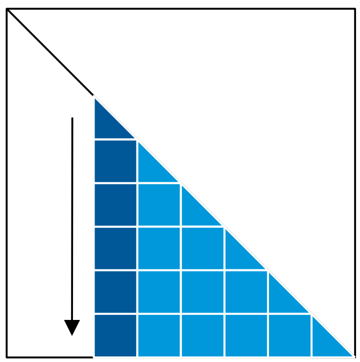
```
for (k = 0; k < A.mt; k++) {
 Insert_Task( zgeqrt,  A[k][k], INOUT,
                       T[k][k], OUTPUT);
 for (m = k+1; m < A.mt; m++) {
  Insert_Task( ztsqrt,  A[k][k], INOUT | REGION_D|REGION_U,
                        A[m][k], INOUT | LOCALITY,
                        T[m][k], OUTPUT);
 }
 for (n = k+1; n < A.nt; n++) {
  Insert_Task( zunmqr,  A[k][k], INPUT | REGION_L,
                        T[k][k], INPUT,
                        A[k][m], INOUT);
  for (m = k+1; m < A.mt; m++) {
   Insert_Task( ztsmqr, A[k][n], INOUT,
                        A[m][n], INOUT | LOCALITY,
                        A[m][k], INPUT,
                        T[m][k], INPUT);
  }
 }
}
```

- Sequential C code
- Annotated through QUARK-specific syntax
  - Insert_Task
  - INOUT, OUTPUT, INPUT
  - REGION_L, REGION_U, REGION_D, …
  - LOCALITY
- Executes thru the QUARK RT to run on multicore SMPs

Algorithms

Cholesky

PLASMA_[scdz]**potrf**[_Tile][_Async]()

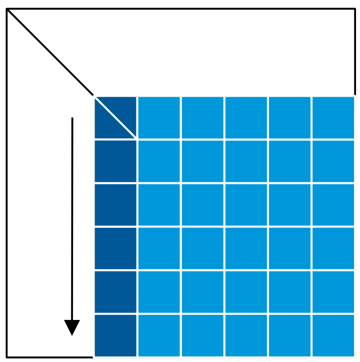


- **Algorithm**
  - equivalent to LAPACK
- **Numerics**
  - same as LAPACK
- **Performance**
  - comparable to vendor on few cores
  - much better than vendor on many cores

Algorithms

LU

PLASMA_[scdz]**getrf**[_Tile][_Async]()

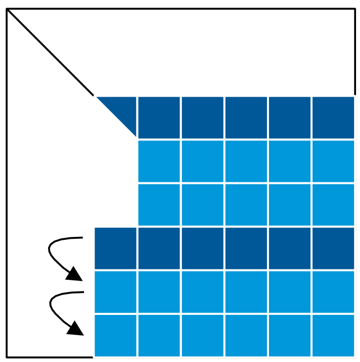


- **Algorithm**
  - equivalent to LAPACK
  - same pivot vector
  - same L and U factors
  - same forward substitution procedure
- **Numerics**
  - same as LAPACK
- **Performance**
  - comparable to vendor on few cores
  - much better than vendor on many cores

Algorithms

incremental QR Factorization

PLASMA_[scdz]**geqrt**[_Tile][_Async]()

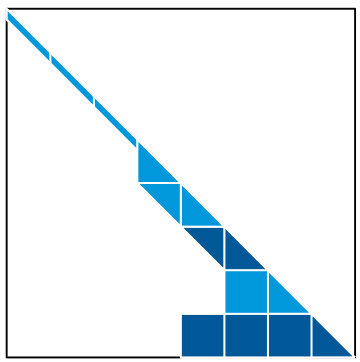


- **Algorithm**
  - the same R factor as LAPACK (absolute values)
  - different set of Householder reflectors
  - different Q matrix
  - different Q generation / application procedure
- **Numerics**
  - same as LAPACK
- **Performance**
  - comparable to vendor on few cores
  - much better than vendor on many cores

Algorithms

three-stage symmetric EVP

PLASMA_[scdz]**syev**[_Tile][_Async]()



- **Algorithm**
  - two-stage tridiagonal reduction + QR Algorithm
  - fast eigenvalues, slower eigenvectors
  (possibility to calculate a subset)

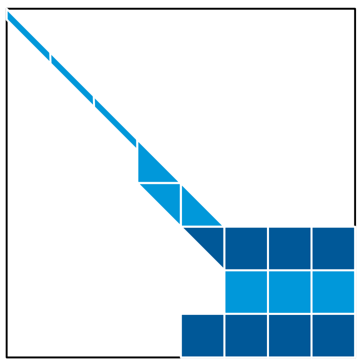
- **Numerics**
  - same as LAPACK

- **Performance**
  - comparable to MKL for very small problems
  - absolutely superior for larger problems

Algorithms

three-stage SVD

PLASMA_[scdz]**gesvd**[_Tile][_Async]()



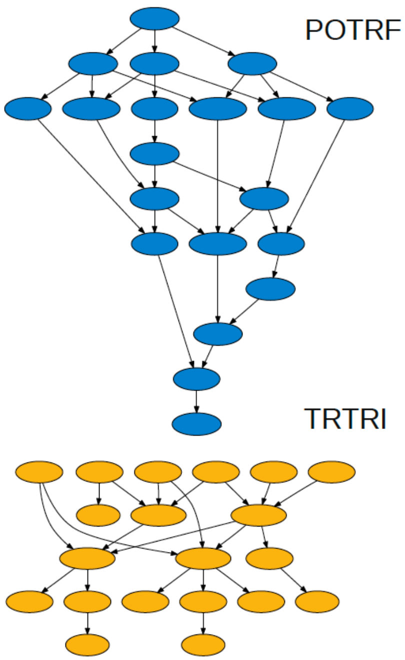
- **Algorithm**
  - two-stage bidiagonal reduction + QR iteration
  - fast singular values, slower singular vectors

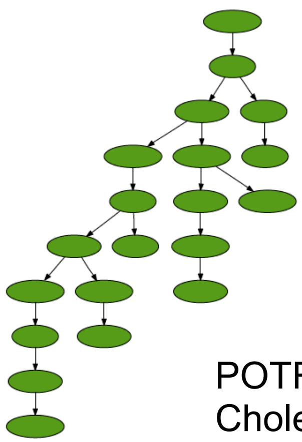
  (possibility of calculating a subset)

- **Numerics**
  - same as LAPACK

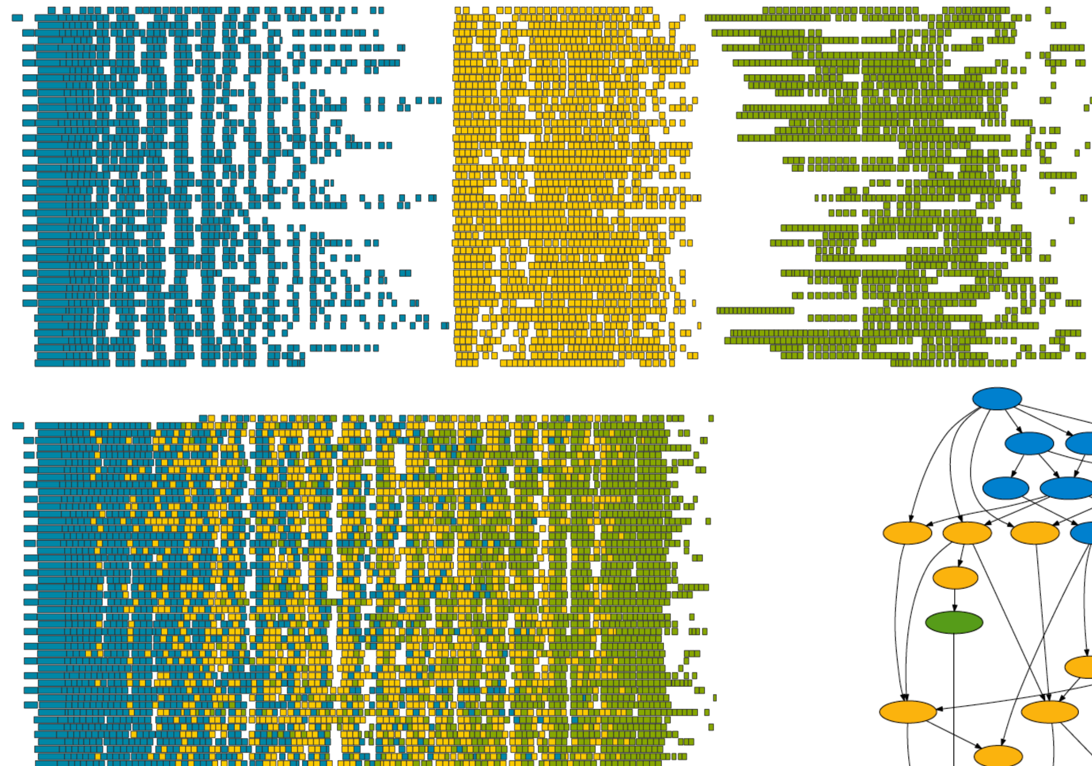
- **Performance**
  - comparable with MKL for very small problems
  - absolutely superior for larger problems

Pipelining: Cholesky Inversion
3 Steps: Factor, Invert L, Multiply L’s

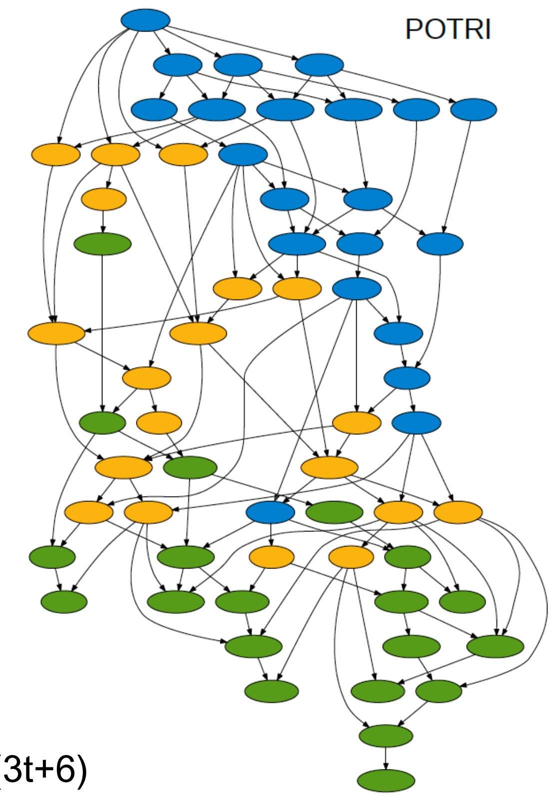




LAUUM



48 cores
POTRF, TRTRI and LAUUM.
The matrix is 4000 x 4000,tile size is 200 x 200,



POTRF+TRTRI+LAUUM: 25 (7t-3)
Cholesky Factorization alone: 3t-2

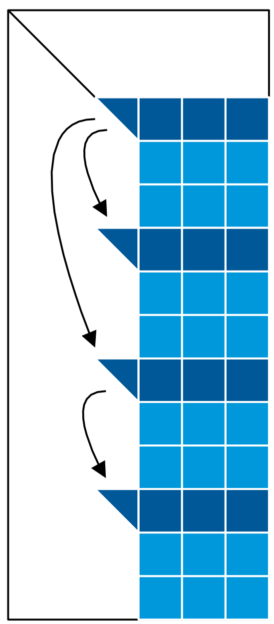
Pipelined: 18 (3t+6)

Algorithms

incremental QR

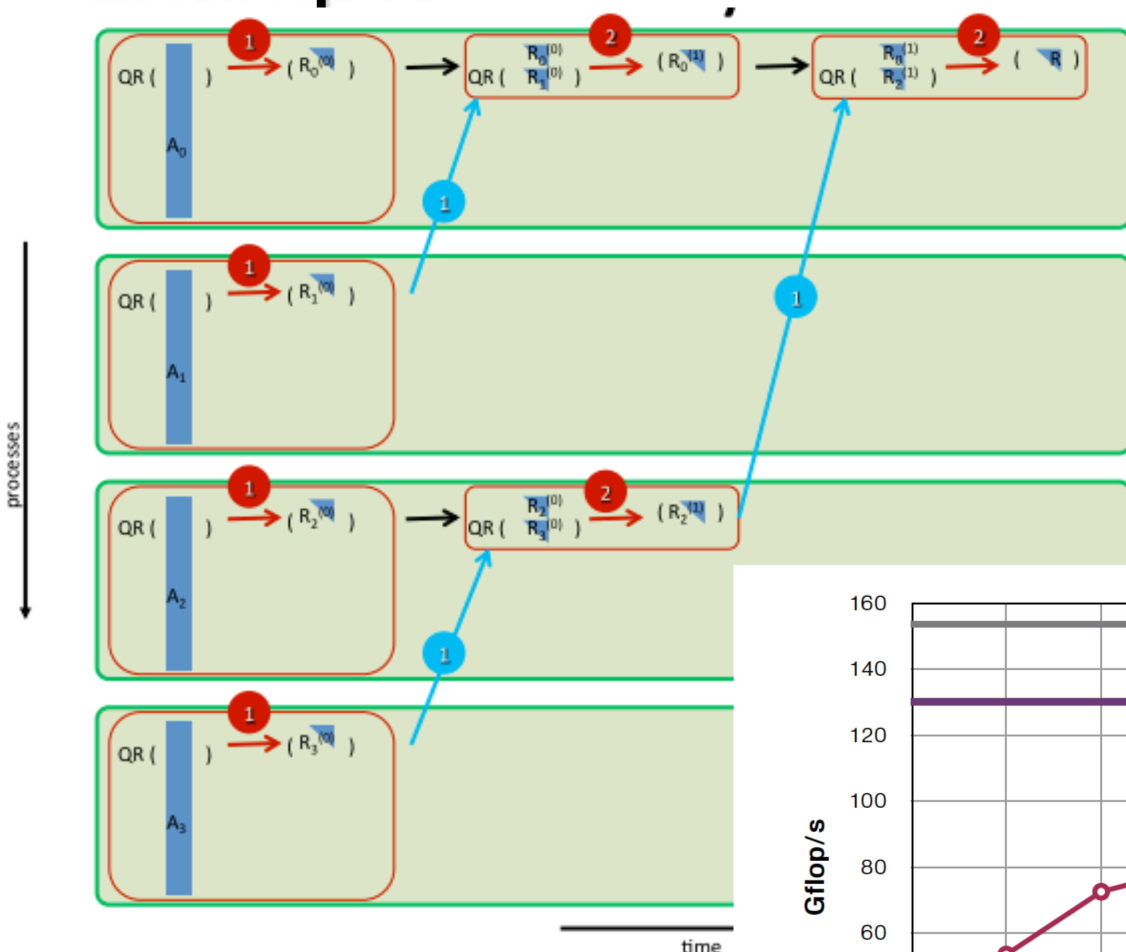
```
PLASMA_[scdz]geqrt[_Tile][_Async]()
```

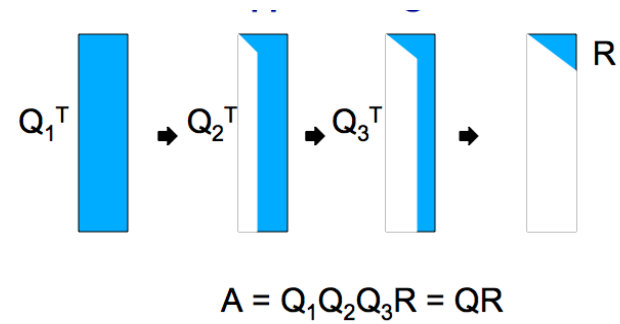
```
PLASMA_Set(
    PLASMA_HOUSEHOLDER_MODE,
    PLASMA_TREE_HOUSEHOLDER);
```



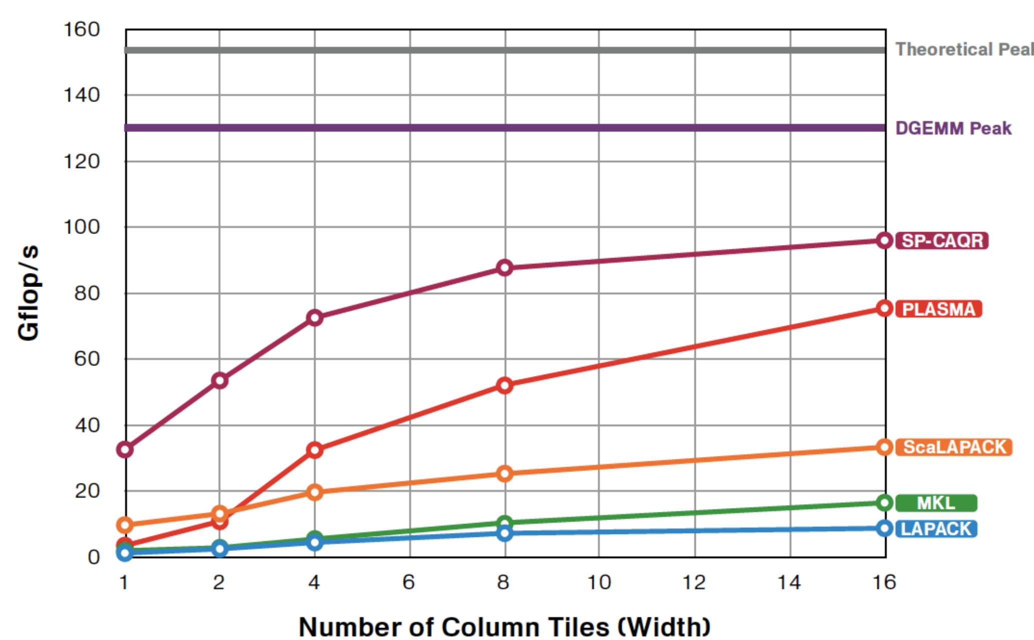
- **Algorithm**
  - the same R factor as LAPACK (absolute values)
  - different set of Householder reflectors
  - different Q matrix
  - different Q generation / application procedure
- **Numerics**
  - same as LAPACK
- **Performance**
  - absolutely superior for tall matrices

Communication Avoiding QR Example





Quad-socket, quad-core machine Intel Xeon EMT64 E7340 at 2.39 GHz.
Theoretical peak is 153.2 Gflop/s with 16 cores.
Matrix size 51200 by 3200



Random Butterfly Pivoting (RBP)

- **To solve $Ax = b$:**
  - Compute $A_r = U^T A V$, with U and V random matrices
  - Factorize A_r without pivoting (GENP)
  - Solve $A_r\, y = U^T b$ and then Solve $x = Vy$
- **U and V are Recursive Butterfly Matrices**
  - Randomization is cheap ($O(n_2)$ operations)
  - GENP is fast ("Cholesky" speed, take advantage of the GPU)
  - Accuracy is in practice similar to GEPP (with iterative refinement), but…

A **butterfly matrix** is defined as any *n*-by-*n* matrix of the form:

$$B = \frac{1}{\sqrt{2}} \begin{pmatrix} R & S \\ R & -S \end{pmatrix}$$

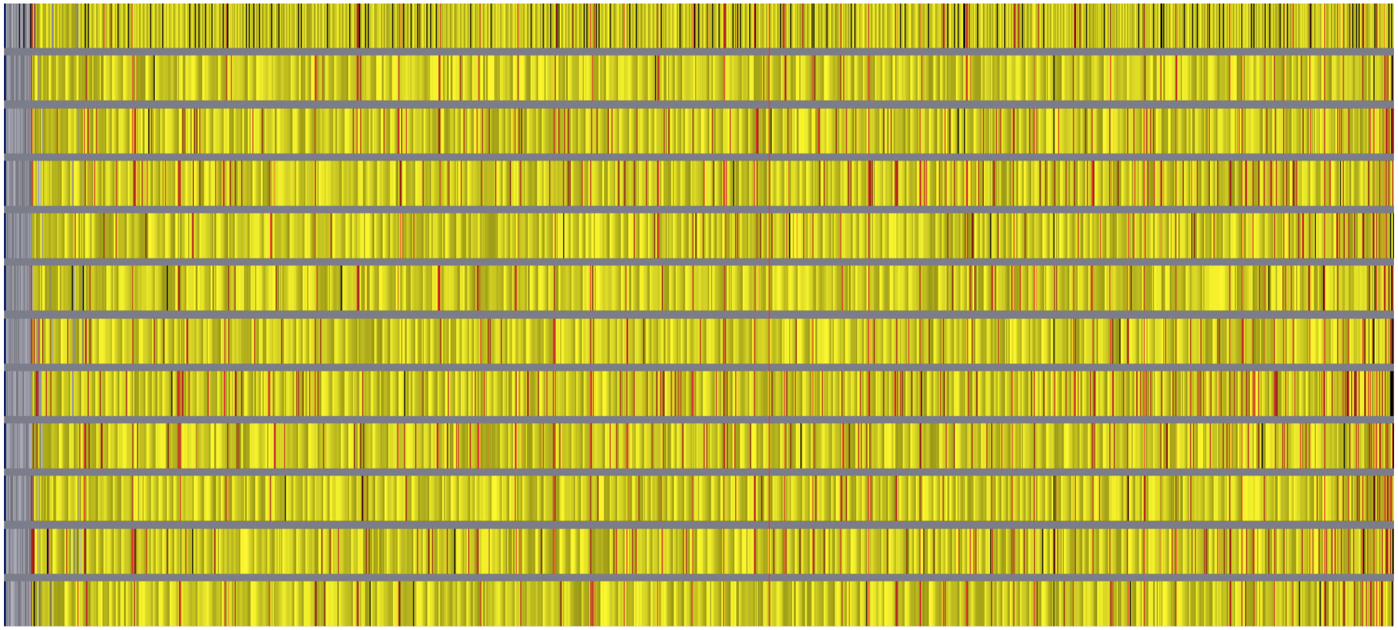
where *R* and *S* are random diagonal matrices.

$B =$ 

Think of this as a preconditioner step.

Goal: Transform A into a matrix that would be sufficiently "random" so that, with a probability close to 1, pivoting is not needed.

PLASMA RBT execution trace



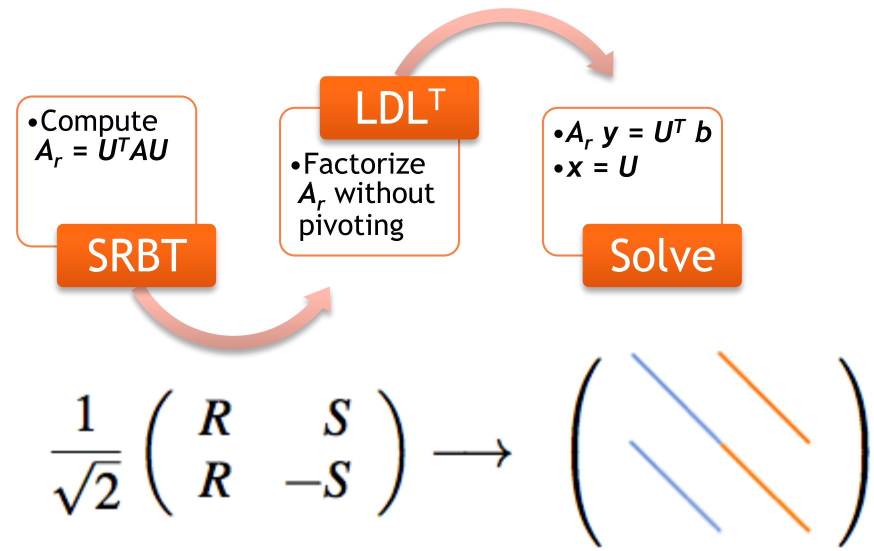
- with n=2000, nb=250 on 12-core AMD Opteron -

- Partial randomization (i.e. gray) is inexpensive.
- Factorization without pivoting is scalable without synchronizations.

Randomize Instead of Pivoting

- *A* is symmetric indefinite. Given the factorization $A = LDL^T$, where L is unit lower triangular and D is diagonal
- Solve $Ax = b$ by solving successively

  $Lz = b, \quad Dy = z, \quad L^Tx = y$
- Not stable
  - To ensure stability usually pivoting is used such as $PAP^T = LDL^T$, where P is a permutation matrix
  - Pivoting complicated and expensive
- Avoid pivoting using Random Butterfly Transformations (RBT)
- Apply iterative refinement to solution
  - If non-convergence call LU on symmetric matrix
- Performance similar to Cholesky



R and S are random diagonal matrices

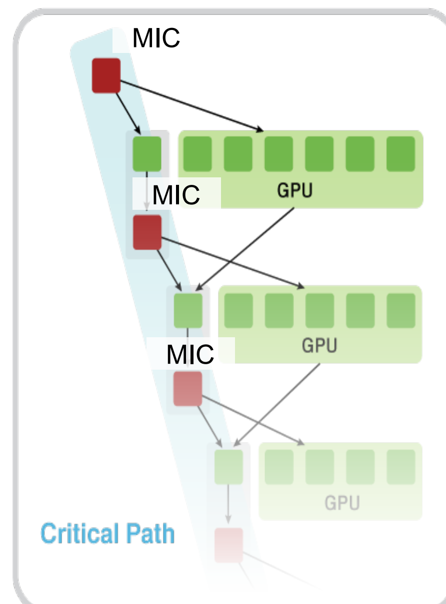
Matrix	Cond A	NP	PP	SRBT (IR)
condex	10^2	10^{-15}	10^{-15}	10^{-15} (0)
fiedler	10^5	–	10^{-15}	10^{-15} (0)
orthog	10^0	10^{-1}	10^{-14}	10^{-16} (1)
randcorr	10^3	10^{-16}	10^{-16}	10^{-16} (0)
augment	10^4	10^{-15}	10^{-15}	10^{-16} (1)
prolate	10^{18}	10^{-15}	10^{-16}	10^{-15} (0)
toeppd	10^7	10^{-16}	10^{-16}	10^{-16} (0)
ris	10^0	–	10^{-15}	10^{-1} (10)
$\|i-j\|$	10^5	10^{-15}	10^{-15}	10^{-14} (0)
max(i,j)	10^6	10^{-14}	10^{-15}	10^{-14} (0)
Hadamard	10^0	10^0	10^0	10^{-15} (0)
rand0	10^5	10^{-12}	10^{-14}	10^{-15} (1)
rand1	10^5	–	10^{-13}	10^{-15} (1)
rand2	10^5	–	10^{-14}	10^{-15} (1)
rand3	10^4	10^{-13}	10^{-14}	10^{-15} (1)

Methodology overview

A methodology to use all available resources:

- MAGMA MIC uses hybridization methodology based on
  - Representing linear algebra algorithms as collections of tasks and data dependencies among them
  - Properly scheduling tasks' execution over multicore CPUs and manycore coprocessors

Hybrid CPU+MIC algorithms (small tasks for multicores and large tasks for MICs)



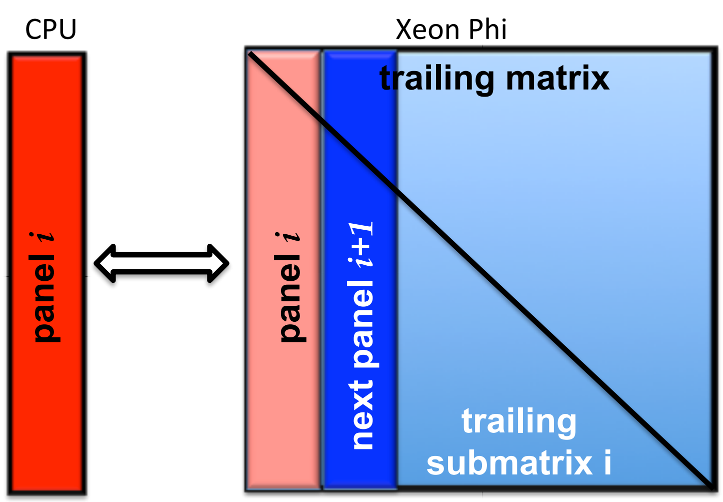
- Successfully applied to fundamental linear algebra algorithms
  - One- and two-sided factorizations and solvers
  - Iterative linear and eigensolvers

- Productivity
  1) High level;
  2) Leveraging prior developments;
  3) Exceeding in performance homogeneous solutions

Hybrid Algorithms

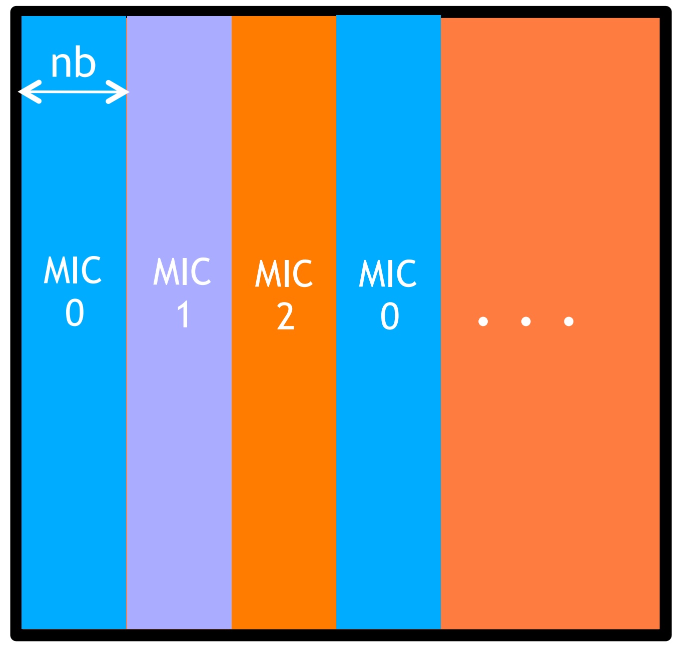
One-Sided Factorizations (LU, QR, and Cholesky)

- **Hybridization**
  - **Panels (Level 2 BLAS) are factored on CPU using LAPACK**
  - **Trailing matrix updates (Level 3 BLAS) are done on the Accelerator using "look-ahead"**

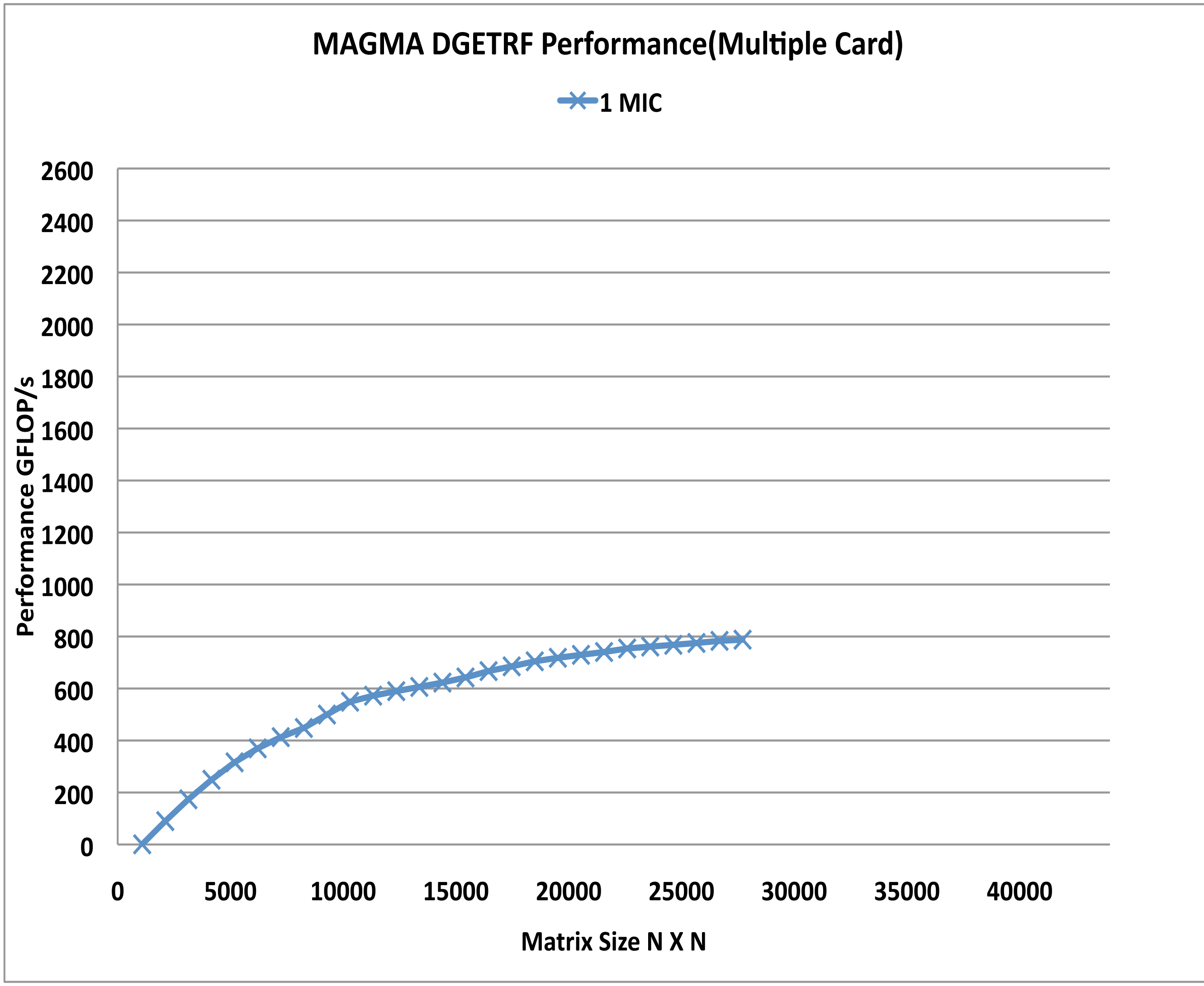


From Single to MultiMIC Support

- Data distribution
  - 1-D block-cyclic distribution
- Algorithm
  - MIC holding current panel is sending it to CPU
  - All updates are done in parallel on the MICs
  - Look-ahead is done with MIC holding the next panel



MAGMA MIC Scalability
LU Factorization Performance in DP

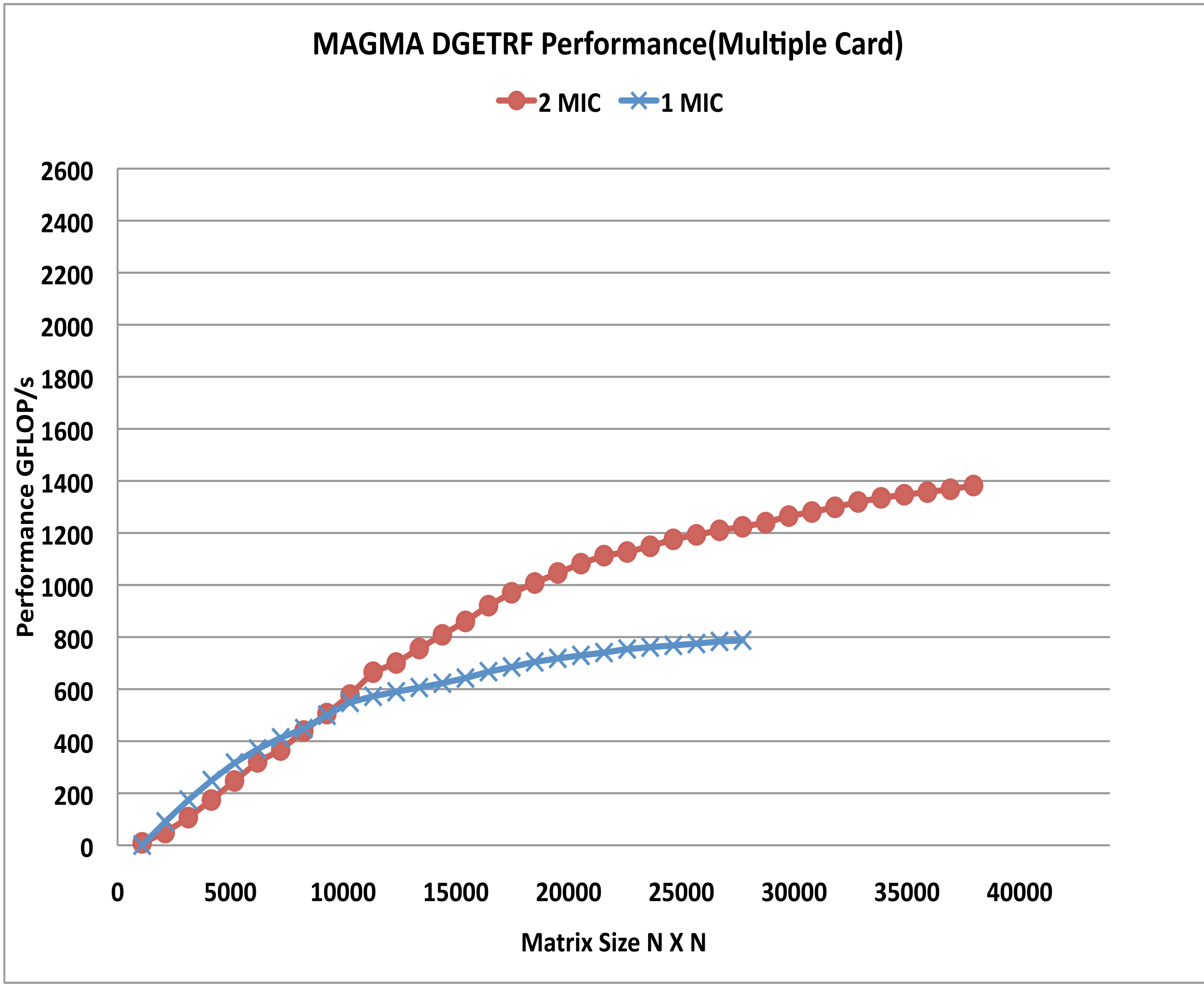


Host
Sandy Bridge (2 x 8 @2.6 GHz)
DP Peak 332 GFlop/s

Coprocessor
Intel Xeon Phi (60 @ 1.09 GHz)
DP Peak 1046 GFlop/s

System DP Peak 1378 GFlop/s
MPSS 2.1.4346-16
compiler_xe_2013.1.117

MAGMA MIC Scalability
LU Factorization Performance in DP

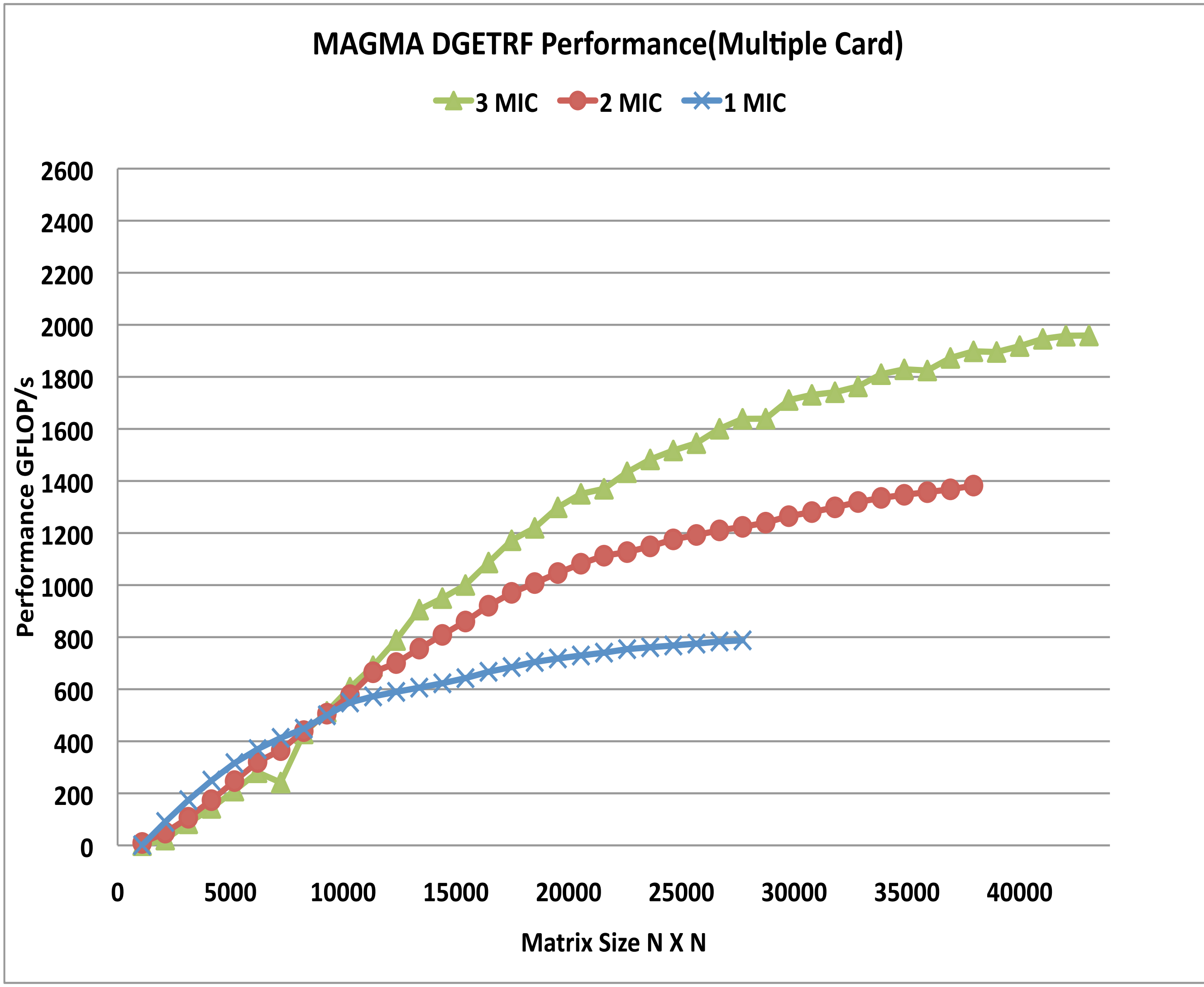


Host
Sandy Bridge (2 x 8 @2.6 GHz)
DP Peak 332 GFlop/s

Coprocessor
Intel Xeon Phi (60 @ 1.09 GHz)
DP Peak 1046 GFlop/s

System DP Peak 1378 GFlop/s
MPSS 2.1.4346-16
compiler_xe_2013.1.117

MAGMA MIC Scalability
LU Factorization Performance in DP

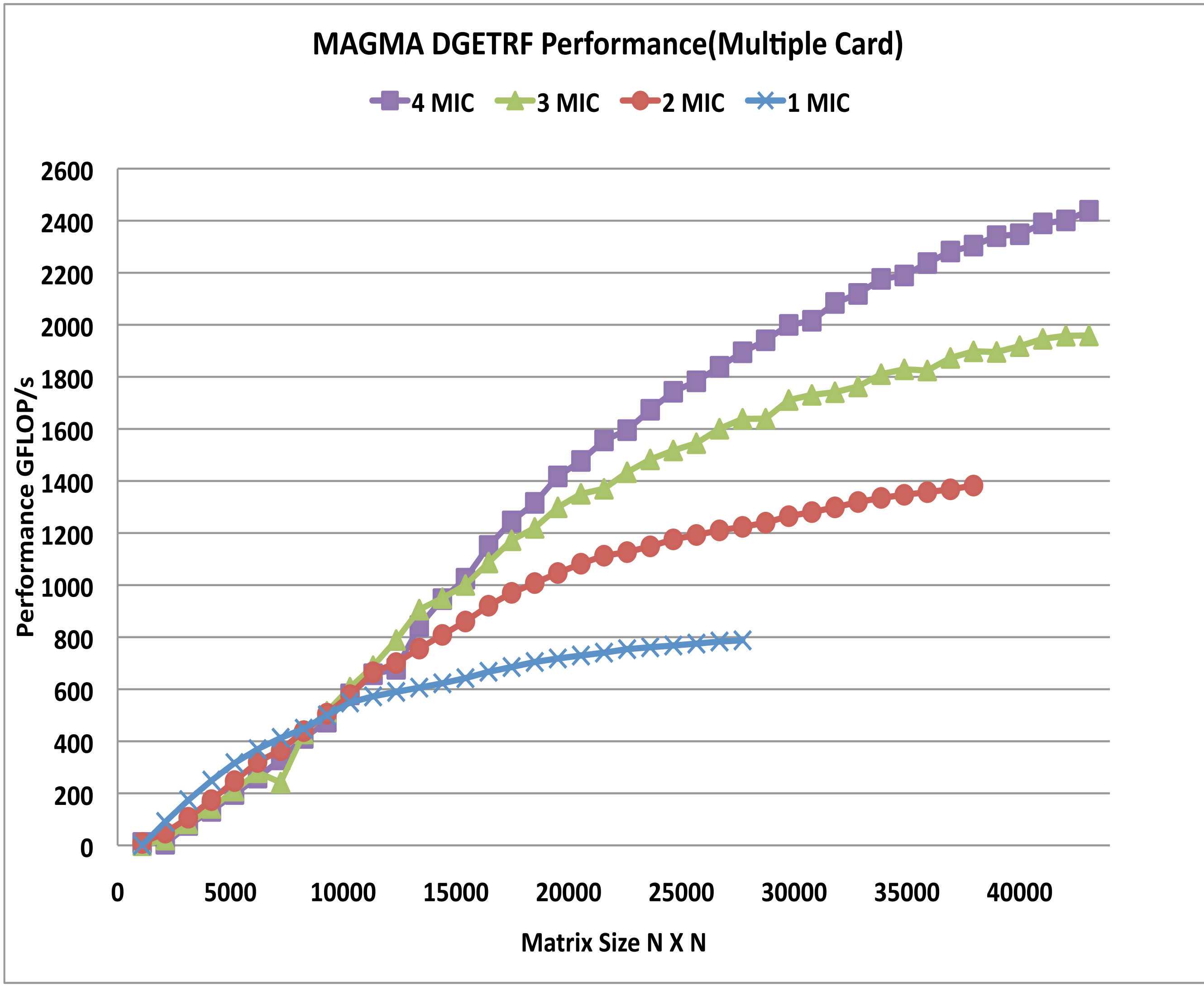


Host
Sandy Bridge (2 x 8 @2.6 GHz)
DP Peak 332 GFlop/s

Coprocessor
Intel Xeon Phi (60 @ 1.09 GHz)
DP Peak 1046 GFlop/s

System DP Peak 1378 GFlop/s
MPSS 2.1.4346-16
compiler_xe_2013.1.117

MAGMA MIC Scalability
LU Factorization Performance in DP



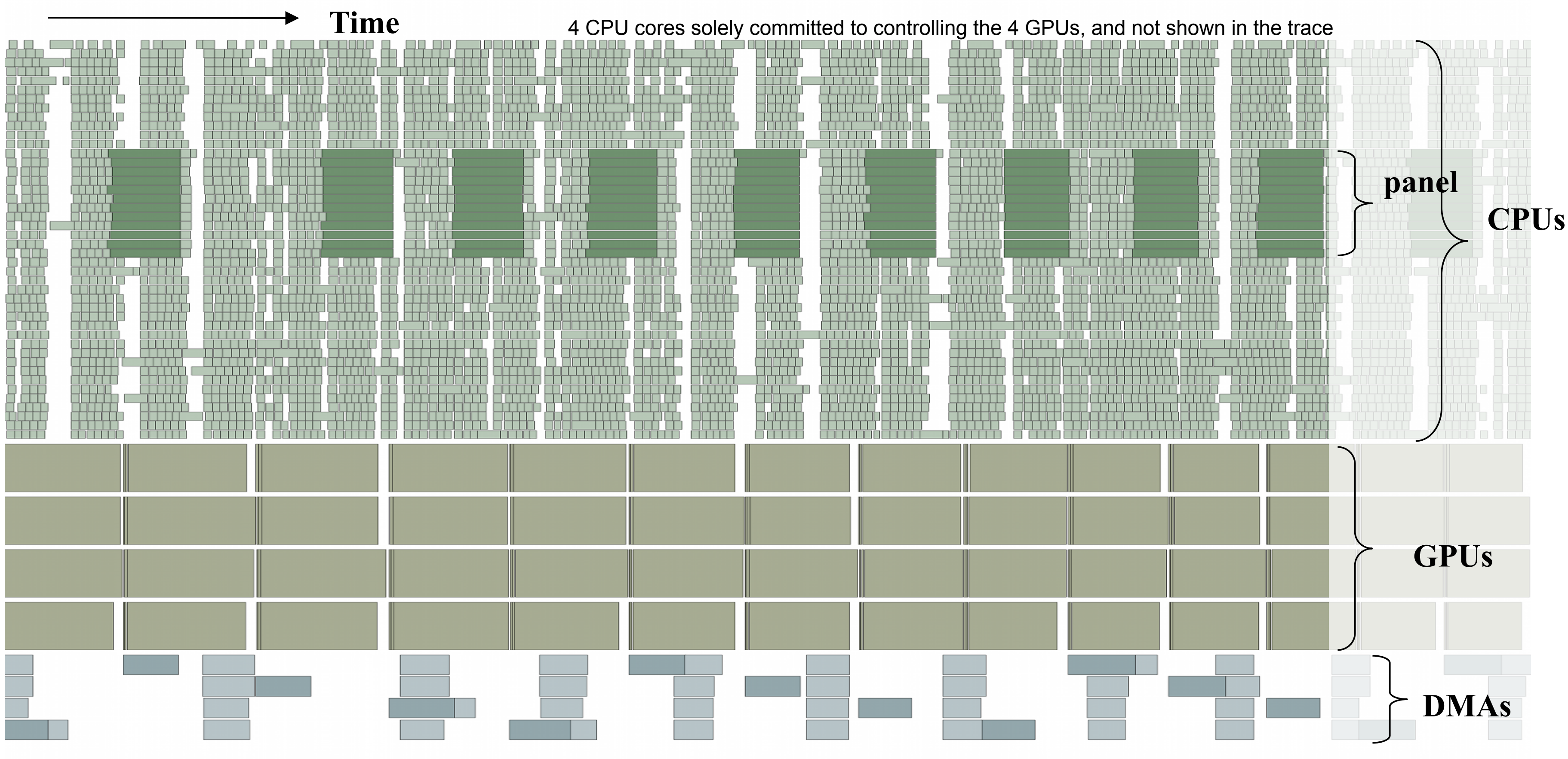
Host
Sandy Bridge (2 x 8 @2.6 GHz)
DP Peak 332 GFlop/s

Coprocessor
Intel Xeon Phi (60 @ 1.09 GHz)
DP Peak 1046 GFlop/s

System DP Peak 1378 GFlop/s
MPSS 2.1.4346-16
compiler_xe_2013.1.117

QUARK on Accelerators

prototype implementation of the LU factorization using 48 cores and 4 GPUs



J. Kurzak, P. Luszczek, M. Faverge, J. Dongarra
Programming the LU Factorization for a Multicore System with Accelerators
High Performance Computing for Computational Science – VECPAR 2012

Mixed Precision Methods

- **Mixed precision, use the lowest precision required to achieve a given accuracy outcome**
  - **Improves runtime, reduce power consumption, lower data movement**
  - **Reformulate to find correction to solution, rather than solution; Δx rather than x.**

$$x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)}$$

$$x_{i+1} - x_i = -\frac{f(x_i)}{f'(x_i)}$$

Idea Goes Something Like This…

- Exploit 32 bit floating point as much as possible.
  - Especially for the bulk of the computation
- Correct or update the solution with selective use of 64 bit floating point to provide a refined results
- Intuitively:
  - Compute a 32 bit result,
  - Calculate a correction to 32 bit result using selected higher precision and,
  - Perform the update of the 32 bit results with the correction using high precision.

Mixed-Precision Iterative Refinement

- Iterative refinement for dense systems, $Ax = b$, can work this way.

L U = lu(A)	$O(n^3)$
x = L\(U\b)	$O(n^2)$
r = b – Ax	$O(n^2)$
WHILE || r || not small enough	
z = L\(U\r)	$O(n^2)$
x = x + z	$O(n^1)$
r = b – Ax	$O(n^2)$
END	

  - Wilkinson, Moler, Stewart, & Higham provide error bound for SP fl pt results when using DP fl pt.

Mixed-Precision Iterative Refinement

- Iterative refinement for dense systems, $Ax = b$, can work this way.

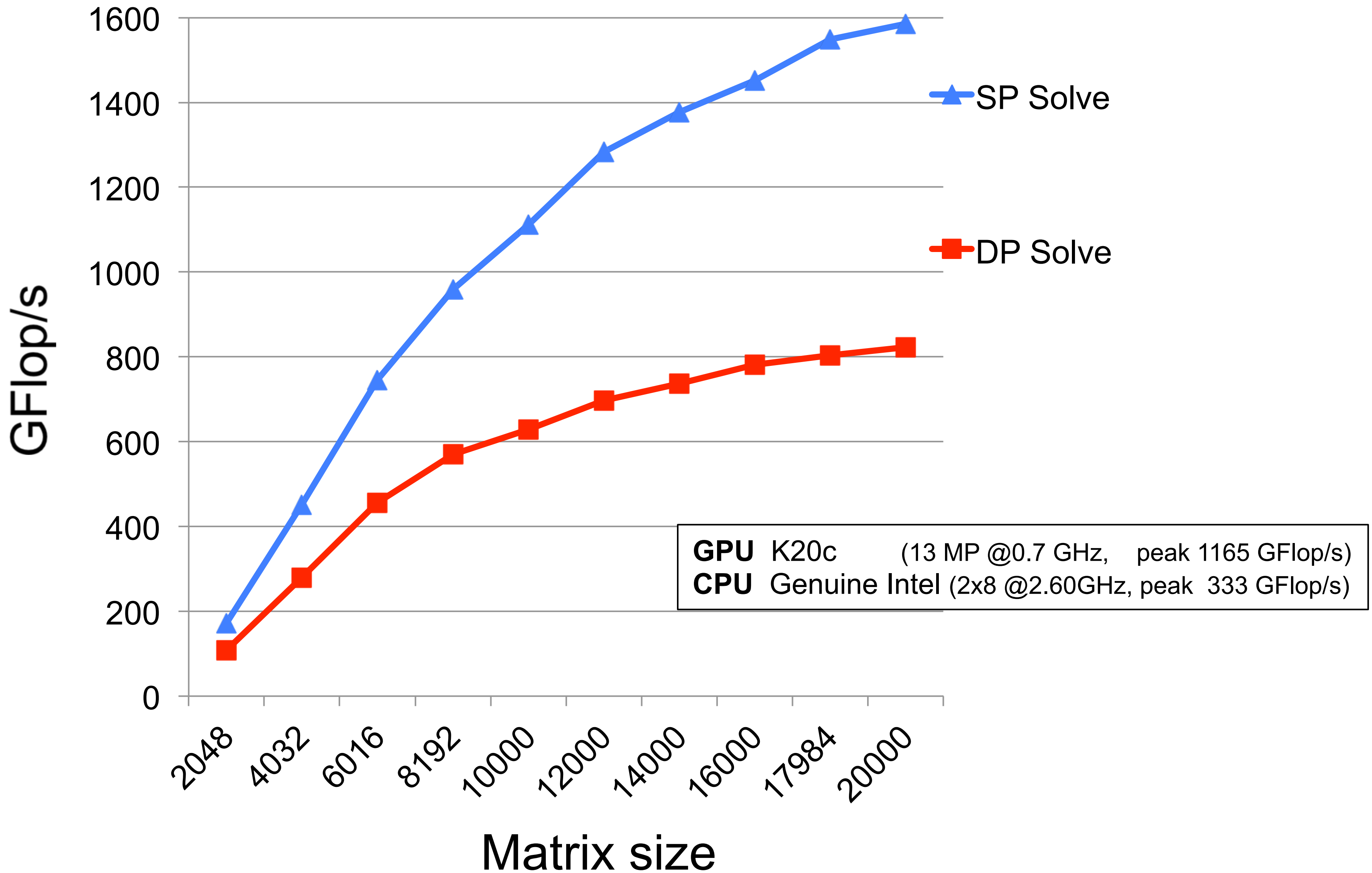
L U = lu(A)	SINGLE	$O(n^3)$
x = L\(U\b)	SINGLE	$O(n^2)$
r = b – Ax	DOUBLE	$O(n^2)$
WHILE || r || not small enough		
z = L\(U\r)	SINGLE	$O(n^2)$
x = x + z	DOUBLE	$O(n^1)$
r = b – Ax	DOUBLE	$O(n^2)$
END		

- Wilkinson, Moler, Stewart, & Higham provide error bound for SP fl pt results when using DP fl pt.
- It can be shown that using this approach we can compute the solution to 64-bit floating point precision.

- Requires extra storage, total is 1.5 times normal;
- $O(n^3)$ work is done in lower precision
- $O(n^2)$ work is done in high precision
- Problems if the matrix is ill-conditioned in sp; $O(10^8)$

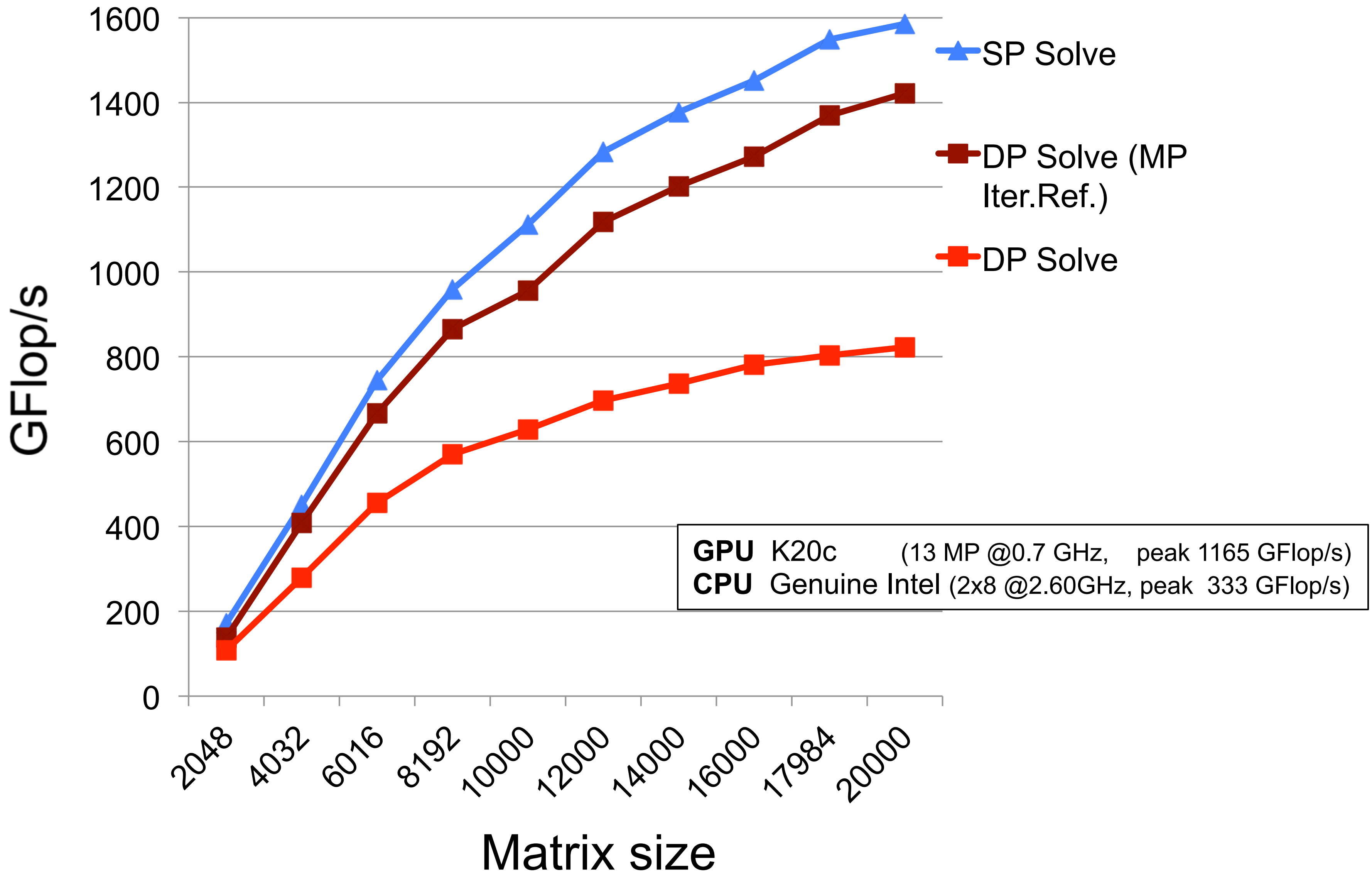
Mixed precision iterative refinement

Solving general dense linear systems using mixed precision iterative refinement

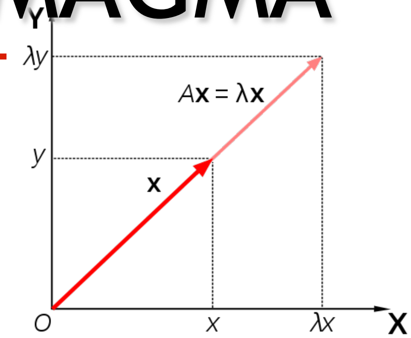


Mixed precision iterative refinement

Solving general dense linear systems using mixed precision iterative refinement

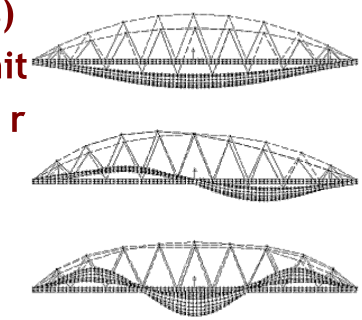


Eigenproblem Solvers in MAGMA



- $A\,x = \lambda\,x$
  - Quantum mechanics (Schrödinger equation)
  - Quantum chemistry
  - Principal component analysis (in data mining)
  - Vibration analysis (of mechanical structures)
  - Image processing, compression, face recognit
  - Eigenvalues of graph, e.g., in Google's page r

    . . .



- **Need to solve it fast**

Current MAGMA results:

MAGMA with 1 GPU can be 12x faster vs vendor libraries on state-of-art multicore systems

T. Dong, J. Dongarra, S. Tomov, I. Yamazaki, T. Schulthess, and R. Solca, *Symmetric dense matrix-vector multiplication on multiple GPUs and its application to symmetric dense and sparse eigenvalue problems*, ICL Technical report, 03/2012.

J. Dongarra, A. Haidar, T. Schulthess, R. Solca, and S. Tomov, *A novel hybrid CPU- GPU generalized eigensolver for electronic structure calculations based on fine grained memory aware tasks*, ICL Technical report, 03/2012.

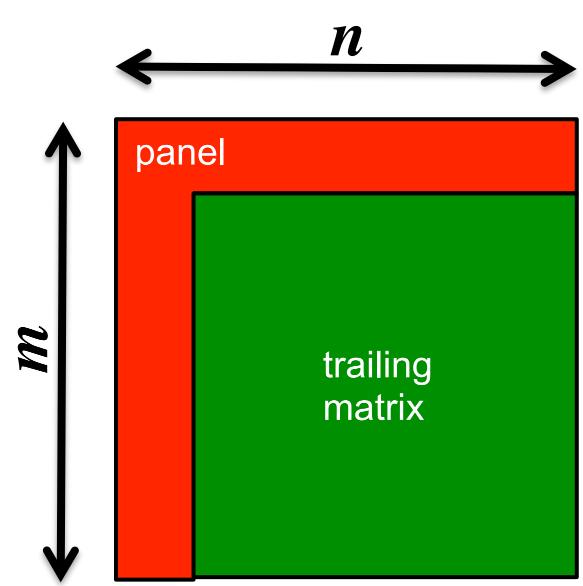
Total Cost of Algorithm

❖For each step it's the cost of the panel + cost of update:

- Each panel is of size nb, and each column of the panel requires:
  - 2 GEMV with the trailing matrix,
  - 6 GEMV with the previous column of the panel,
  - 6 GEMV with the previous row of the panel,
  - 2 LARFG and 2 SCAL.

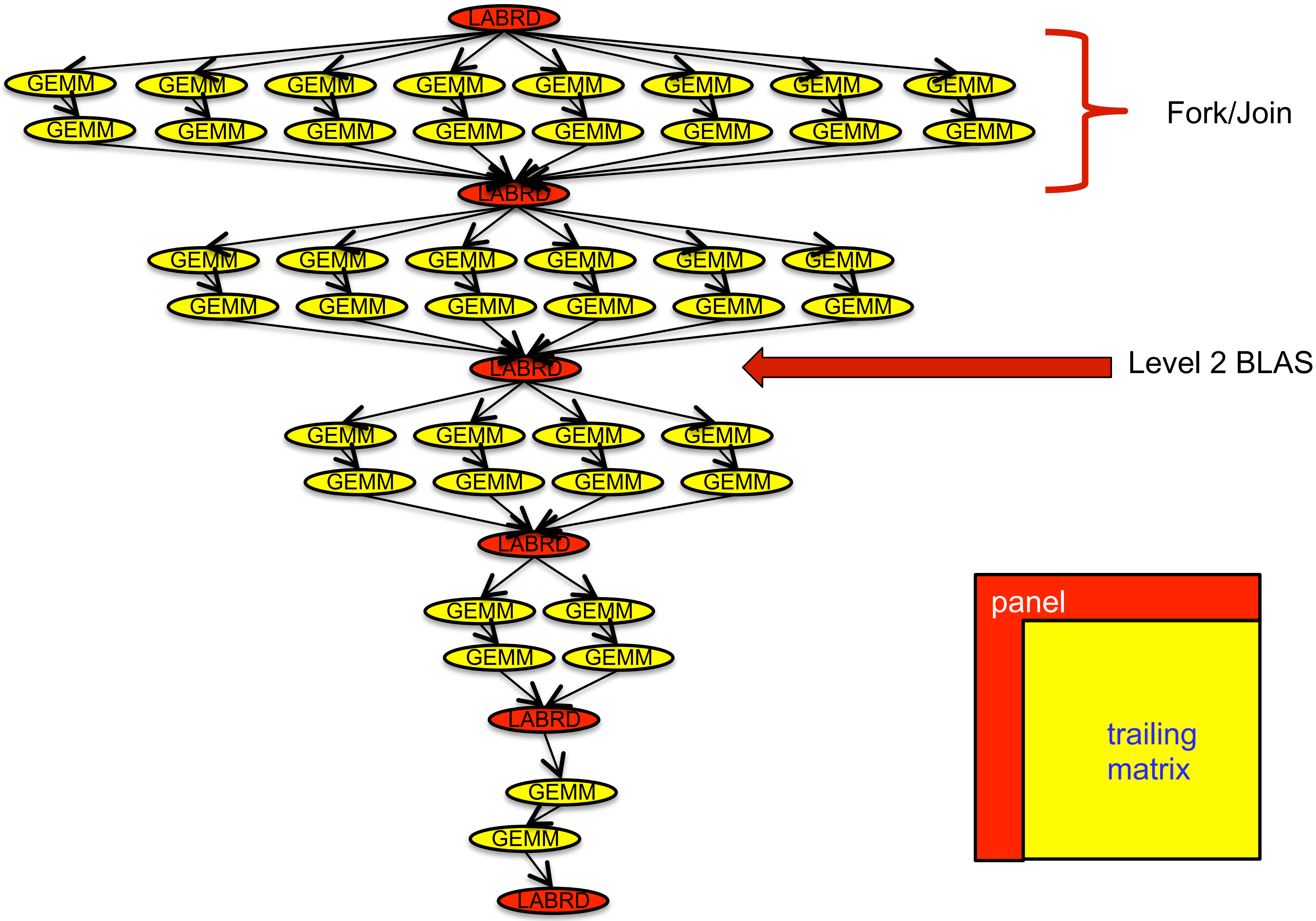
- Thus the cost of a panel is:
  - $nb*(2*2*m*n) + 6*m*nb^2 + 6*n*nb^2 + O(n)$.

- The update **A := A - V*Y' - X*U'** consists into:
  - 2 GEMM of the computed panel to update
    the trailing matrix and so its cost is
    $= 2*(m-nb)*(n-nb)*nb + 2*(m-nb)*(n-nb)*nb$
    $= 4*(m-nb)*(n-nb)*nb$



LARFG Generates an elementary reflector (Householder matrix).

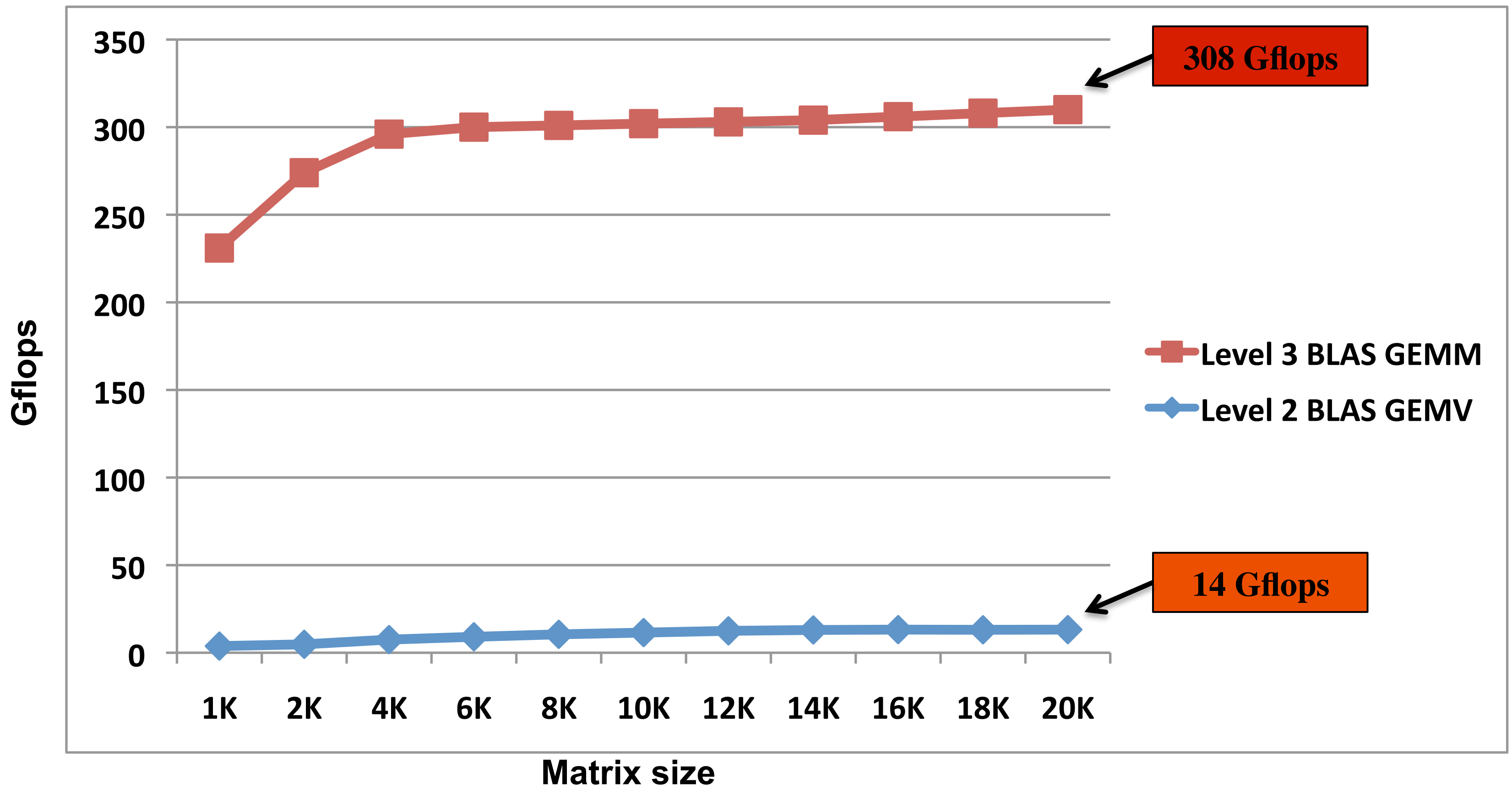
DAG for Conventional Reduction



LABRD: Reduces the first *nb* rows and columns of a general matrix to a bidiagonal form.

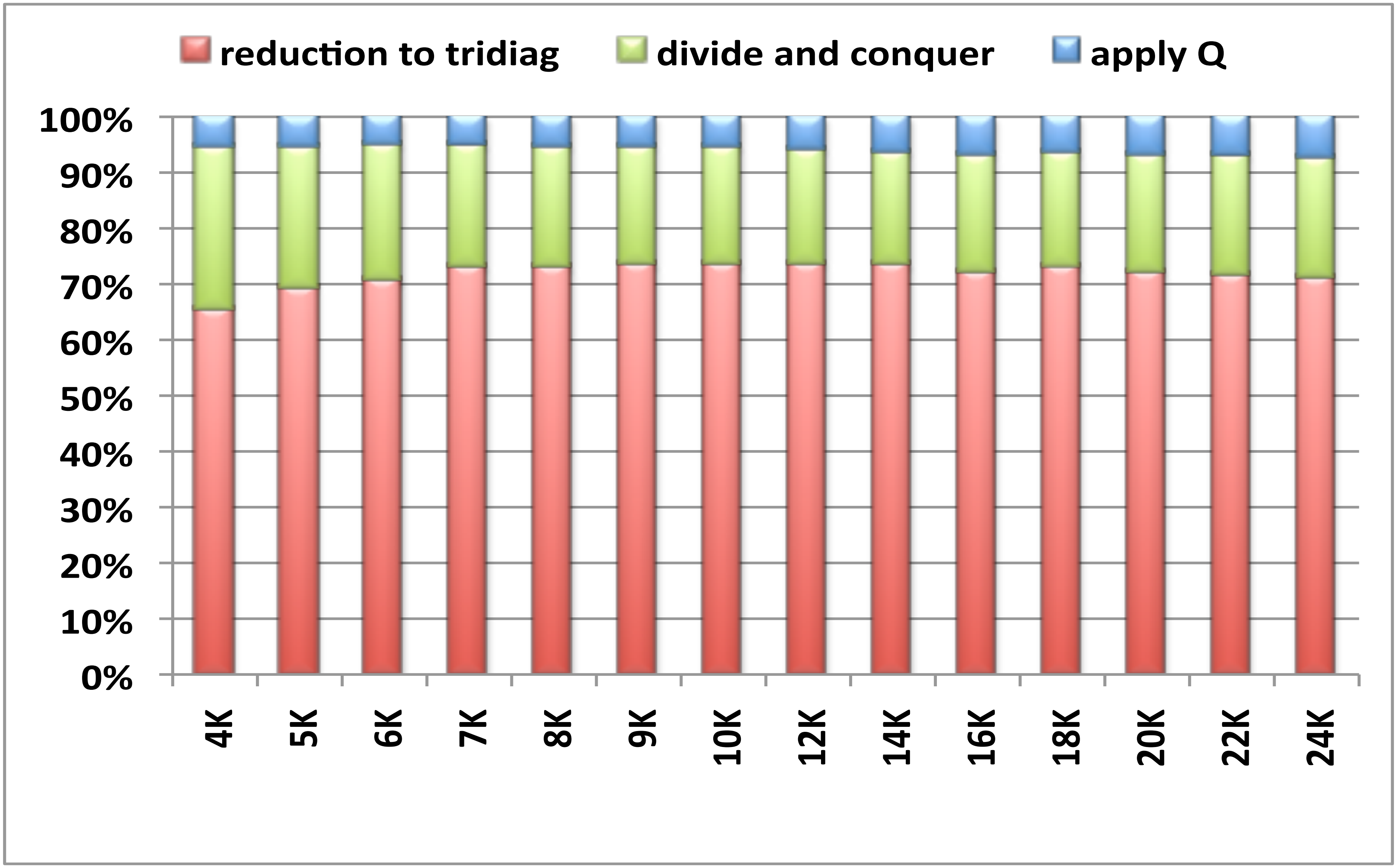
Performance of Level 2 and Level 3 BLAS

❖2 - 8 cores Intel Xeon E5-2670 (Sandy Bridge), 2.6 GHz.
24 MB shared L3 cache, and each core has a private 256 KB L2 and 64 KB L1.
Theoretical peak for this architecture in double precision is 20.8 Gflop/s per core (333 Gflops total).
8 flop/cycle*2.6 cycle/sec*16 cores = 332.8 Gflop/s
Compiled with gcc 4.4.6 and using MKL_composer_xe_2013.3.163



The standard Tridiagonal reduction xSYTRD

The percentage of the time spent in each kernel of the DSYEVDsolver

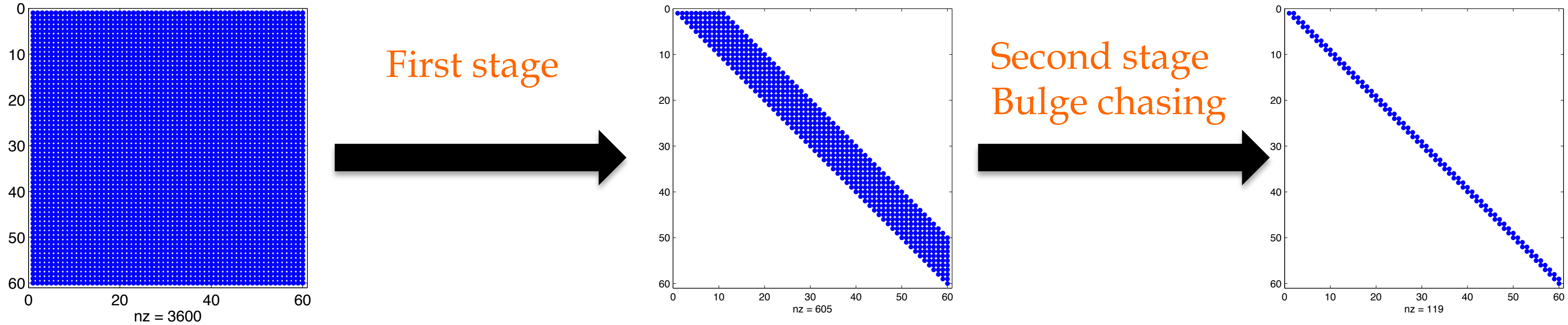


The PLASMA reduction: 2 stage algorithm

Idea:

- The idea is to cast expensive memory operations, occurring during the panel factorization into fast computationally intensive ones.
- Redesign the algorithm in a way which increase the cache reuse. Call it communication reducing.
- Design new cache friendly kernels to overcomes the memory bound limitation.
- Extract parallelism and schedule task in an asynchronous order.

The PLASMA reduction: 2 stage algorithm

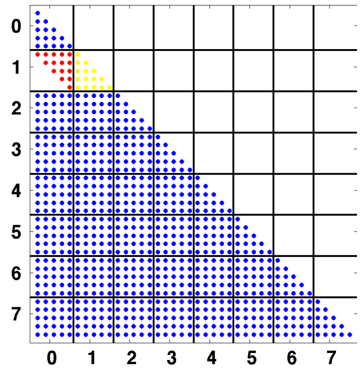


✸ Characteristics

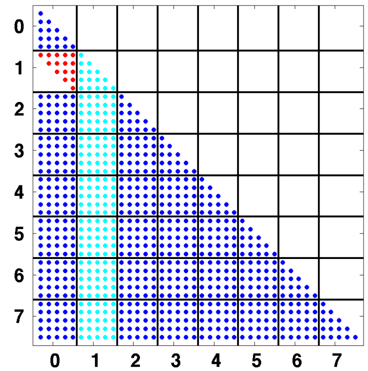
- **Stage 1:**
  - BLAS-3,
  - asynchronous execution,

- **Stage2:**
  - BLAS-1.5,
  - asynchronous execution,
  - new cache friendly kernel (reduced communication).

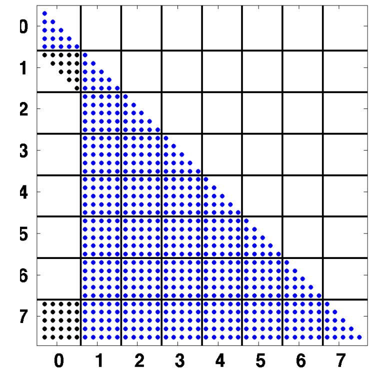
The PLASMA Reduction: 1st Stage



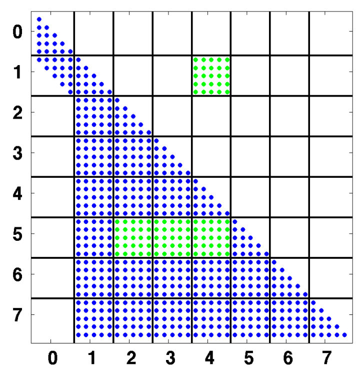
Apply Q from LEFT on tile A(1,1)



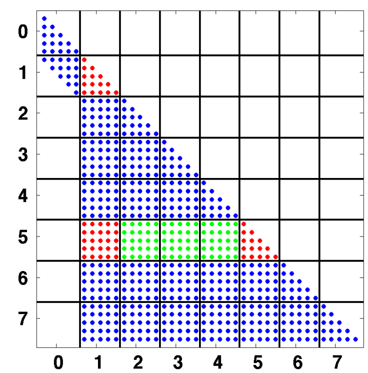
Apply Q from RIGHT on tile A(7,1)



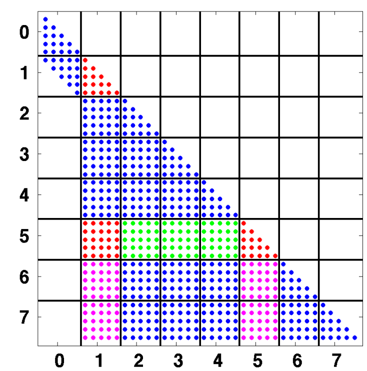
QR factorisation on tile A(1,0) A(7,0)



Apply Q from LEFT on tile $A(4,1)^T$ A(5,4)



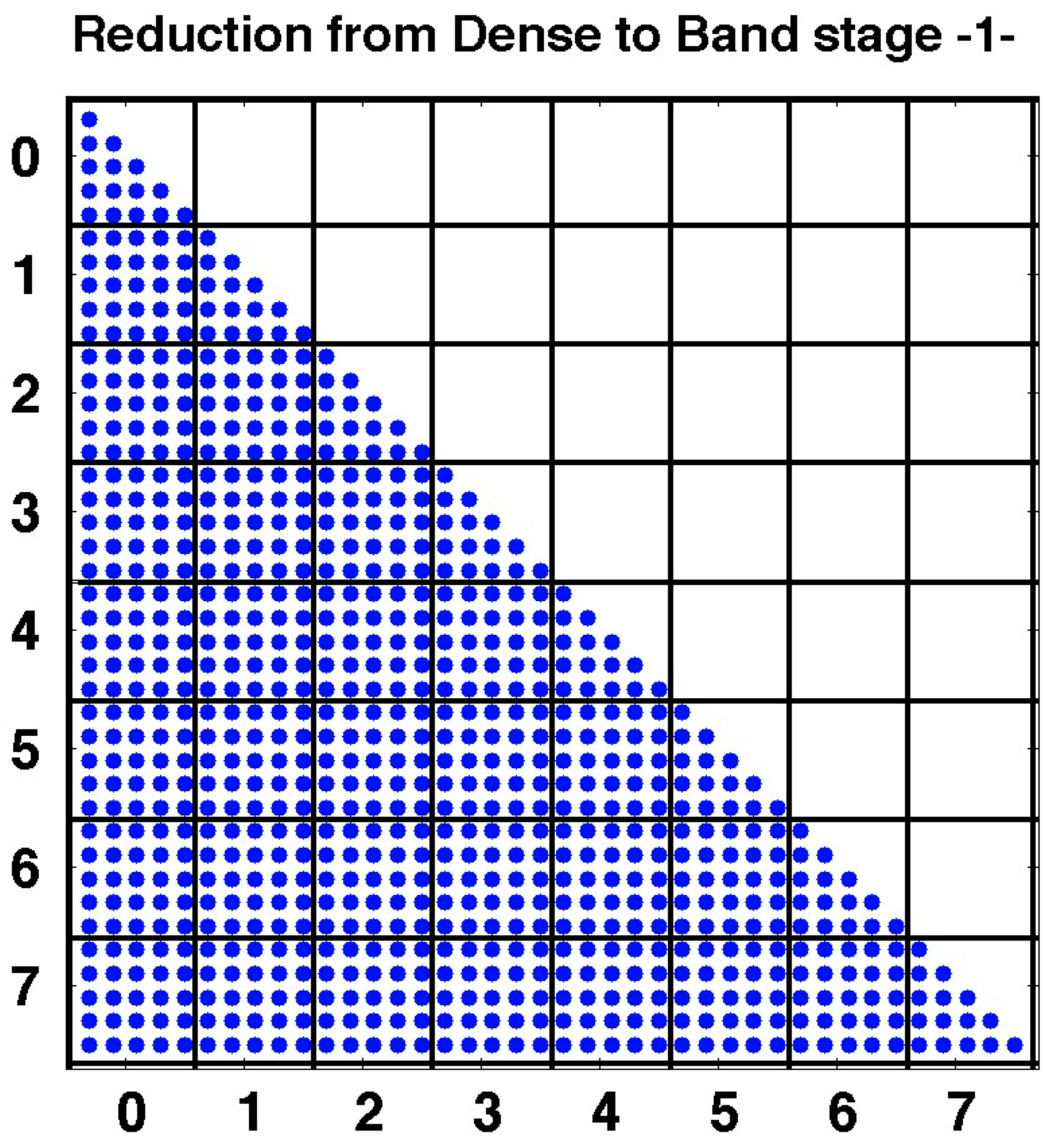
Apply Q from LEFT and RIGHT on tiles A(1,1) A(5,1) A(5,5)



Apply Q from RIGHT on tile A(7,1) A(7,5)

```
for step = 1; 2 to NT-1
    QR factorize
    apply Q from LEFT
    for i = step+1 to NT
        apply Q from RIGHT
    end for
    for k = step+2 to NT do
        factorize 2 tiles
        for j = step+2 to k-1
            LEFT update on 2 tiles
        end for
        apply a LEFT and
        RIGHT update diagonal
        for m = k+1 to NT do
            RIGHT updates
        end for
    end for
end for
```

The PLASMA Reduction: 1st Stage



The PLASMA reduction: 2 stage algorithm

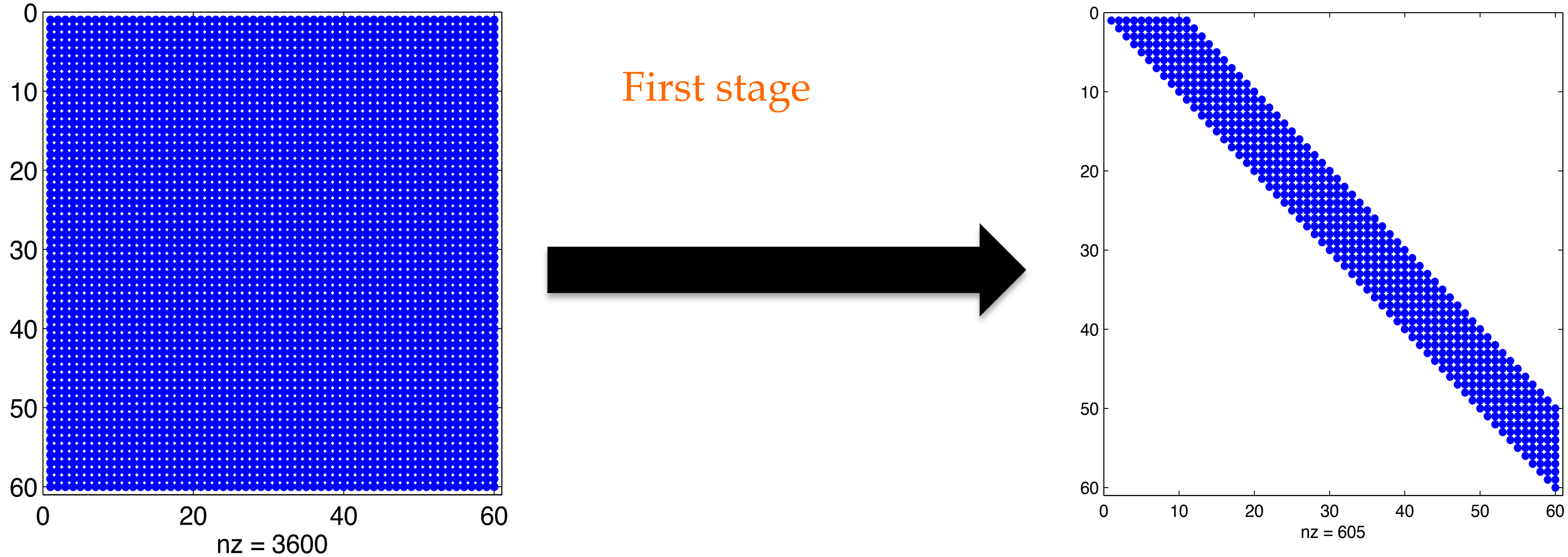
- A. Haidar, P. Luszczek, J. Kurzak and J. Dongarra.
  An Improved Parallel Singular Value Algorithm and Its Implementation for Multicore Hardware.
  International Conference for High Performance Computing, Networking, Storage and Analysis,
  IEEE-SC 2013.

- A. Haidar, R. Solca, M. Gates, S. Tomov, T. Schulthess and J. Dongarra.
  Leading edge multi-GPU algorithms for generalized eigenproblems for electronic structure calculations.
  International Supercomputing Conference IEEE-ISC 2013.

- A. Haidar, H. Ltaief, P. Luszczek and J. Dongarra.
  A Comprehensive Study of Task Coalescing for Selecting Parallelism Granularity in a Two-Stage Bidiagonal Reduction A Comprehensive Study of Task Coalescing for Selecting Parallelism Granularity in a Two-Stage Bidiagonal Reduction. IEEE IPDPS 2012

- A. Haidar, H. Ltaief and J. Dongarra.
  Parallel Memory-Aware Fine-Grained Reduction to Condensed Forms for Symmetric Eigenvalue Problems.
  International Conference for High Performance Computing, Networking, Storage and Analysis,
  IEEE-SC 2011.

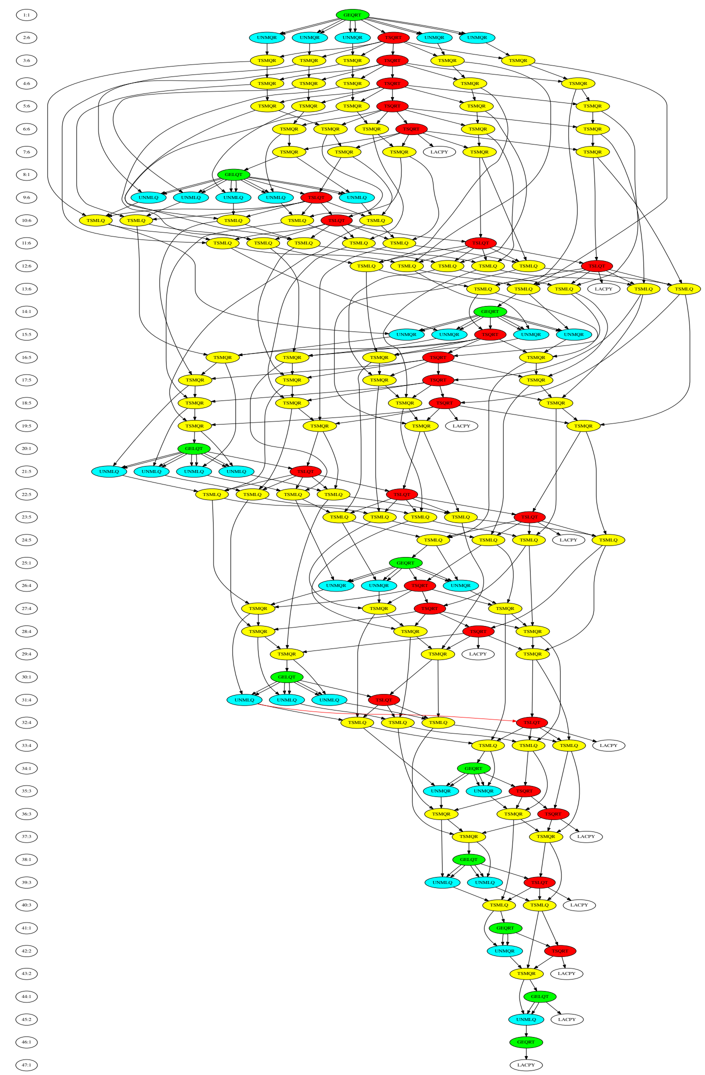
The PLASMA reduction: stage 1



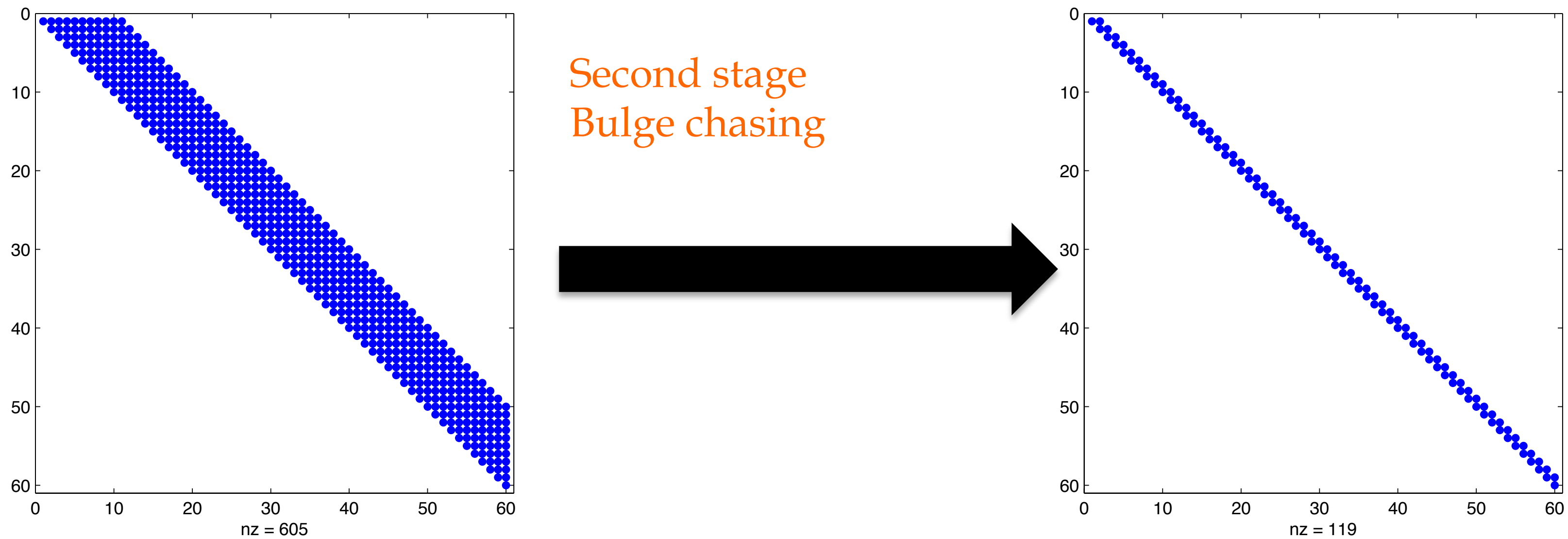
- The algorithm proceeds as a collection of interdependent tasks that operate on the tile data layout.
- These tasks are organized into a directed acyclic graph (DAG) that is executed in an asynchronous manner.

DAG of Stage 1 of 2 Stage Approach

- **Exposes more parallelism**
- **Asynchronous ops**
- **Rich in GEMM**

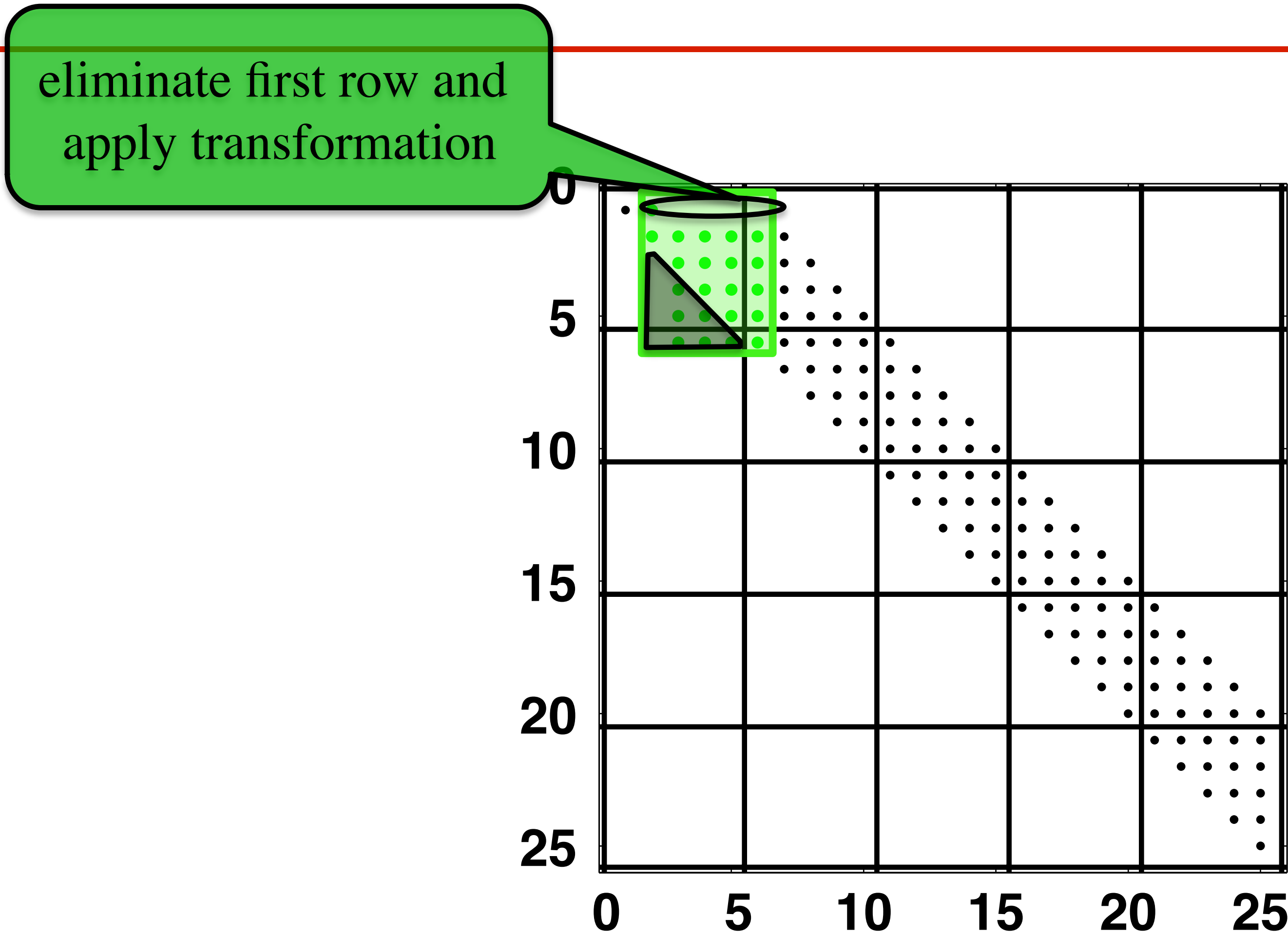


The PLASMA Reduction: 2^{nd} Stage

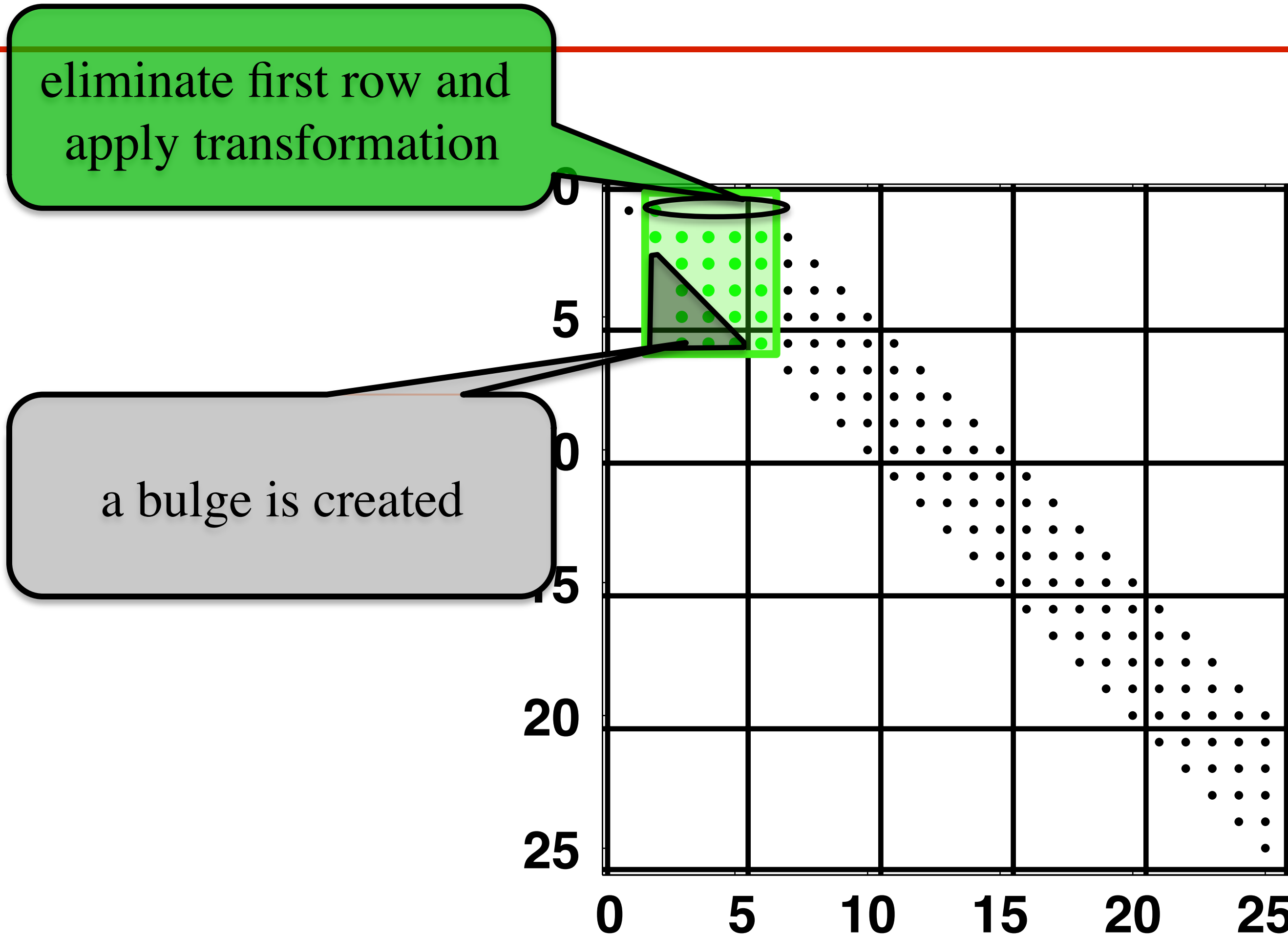


- New cache friendly kernels to overcomes the memory.
- Extract pipelined parallelism and schedule task in order to increase cache reuse.

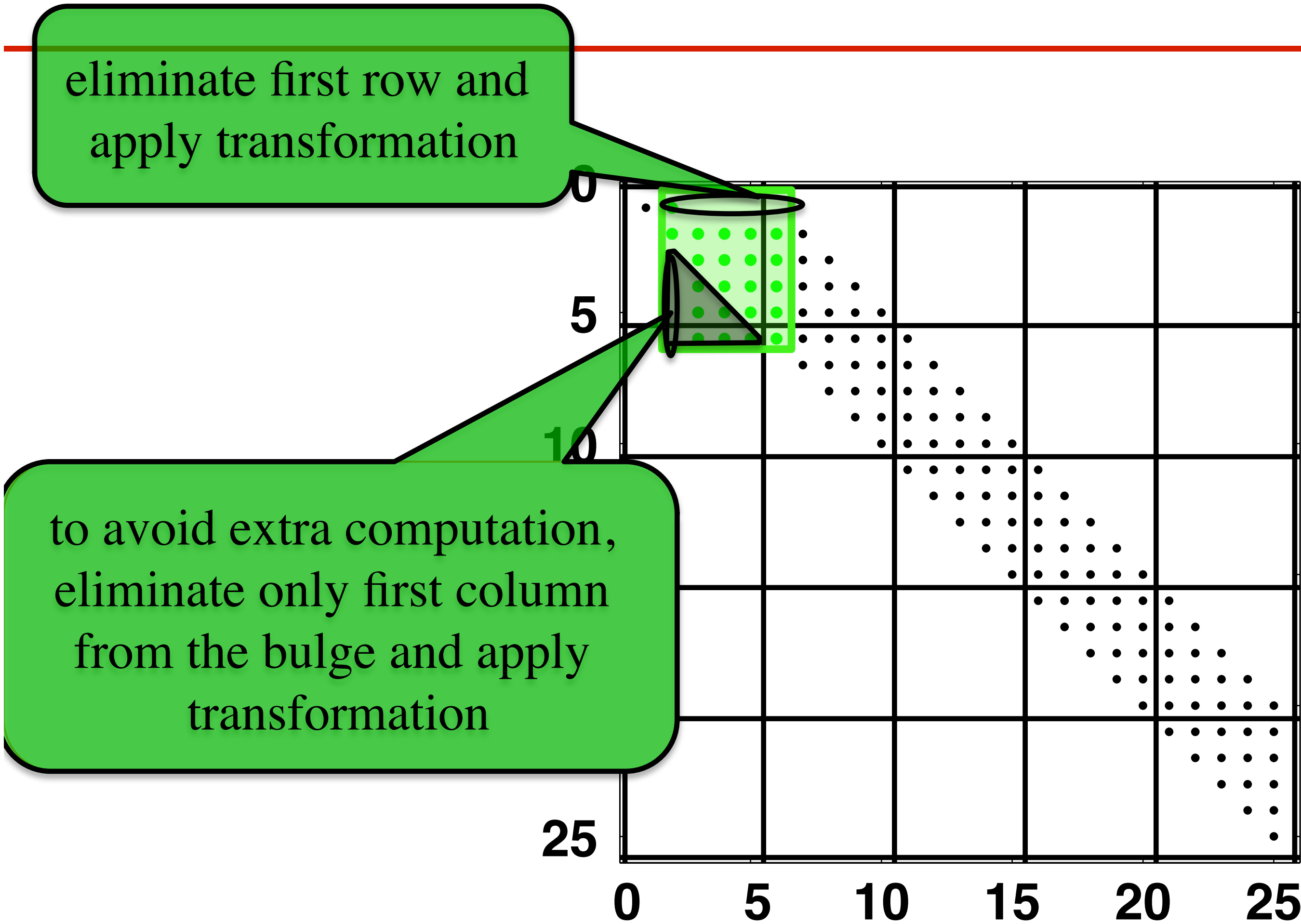
The PLASMA reduction: stage 2



The PLASMA reduction: stage 2

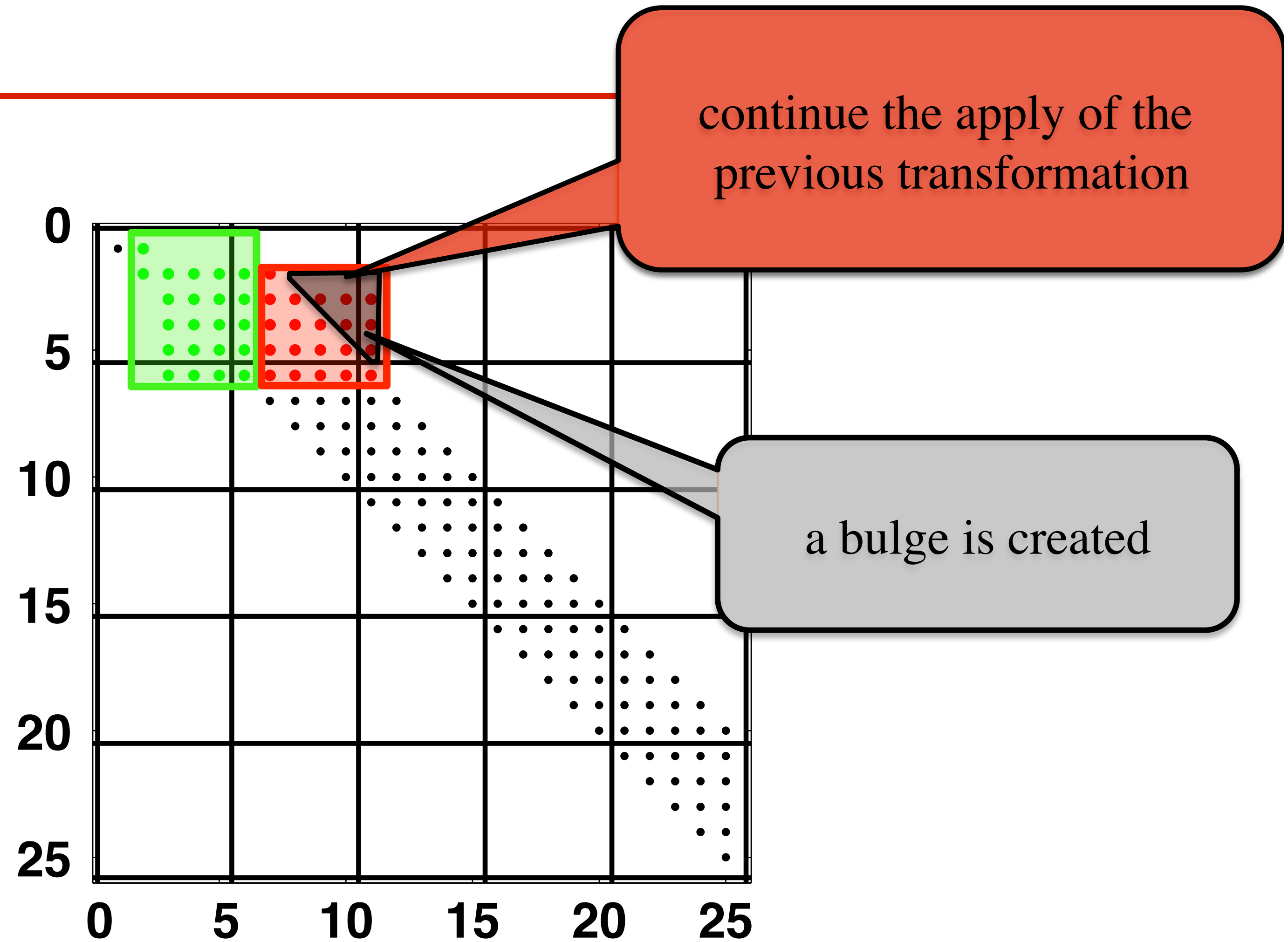


The PLASMA reduction: stage 2

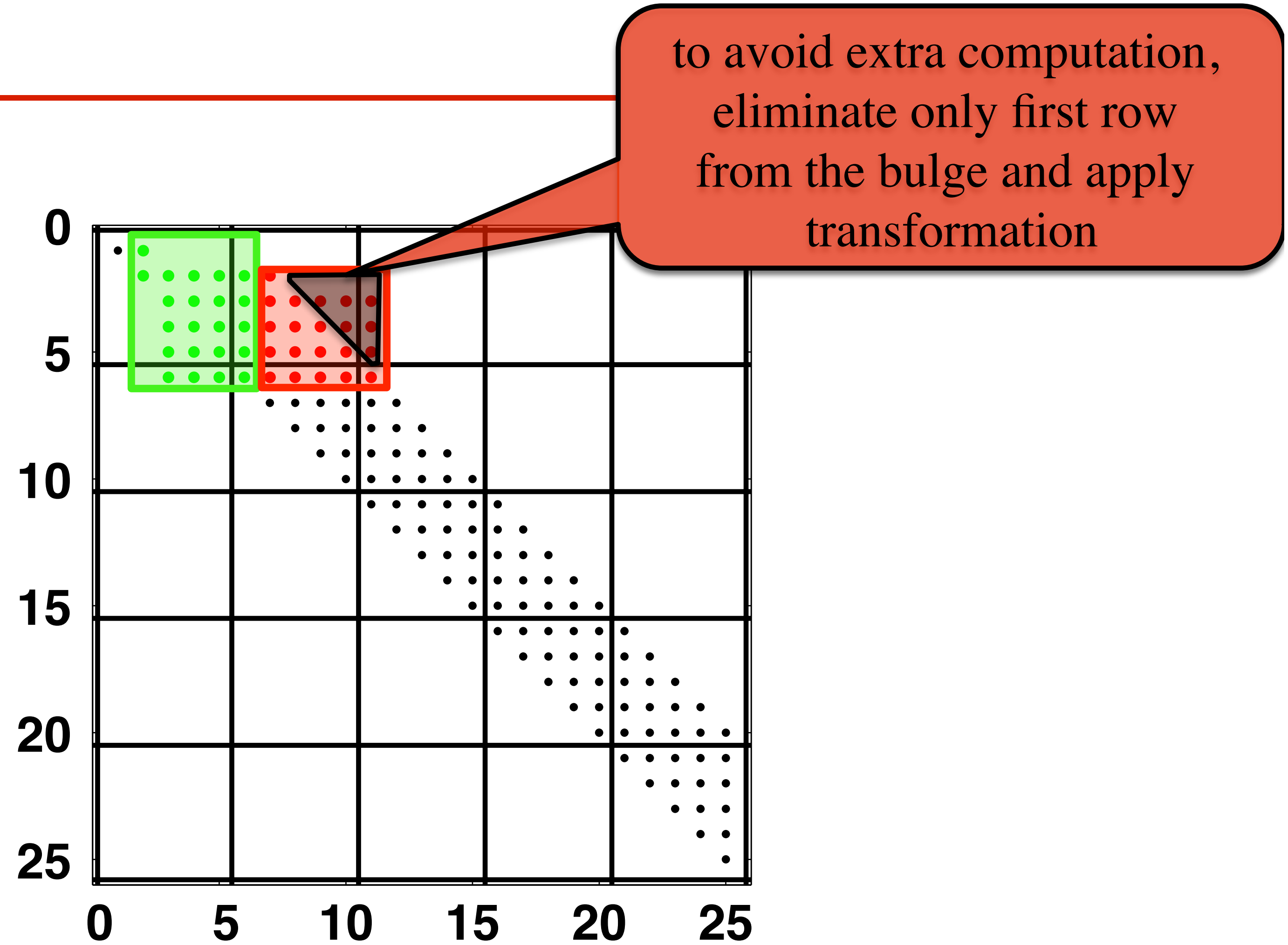


- since the green block of data is small (nbxnb) and to increase cache reuse all of these operations are unrolled within one kernel

The PLASMA reduction: stage 2

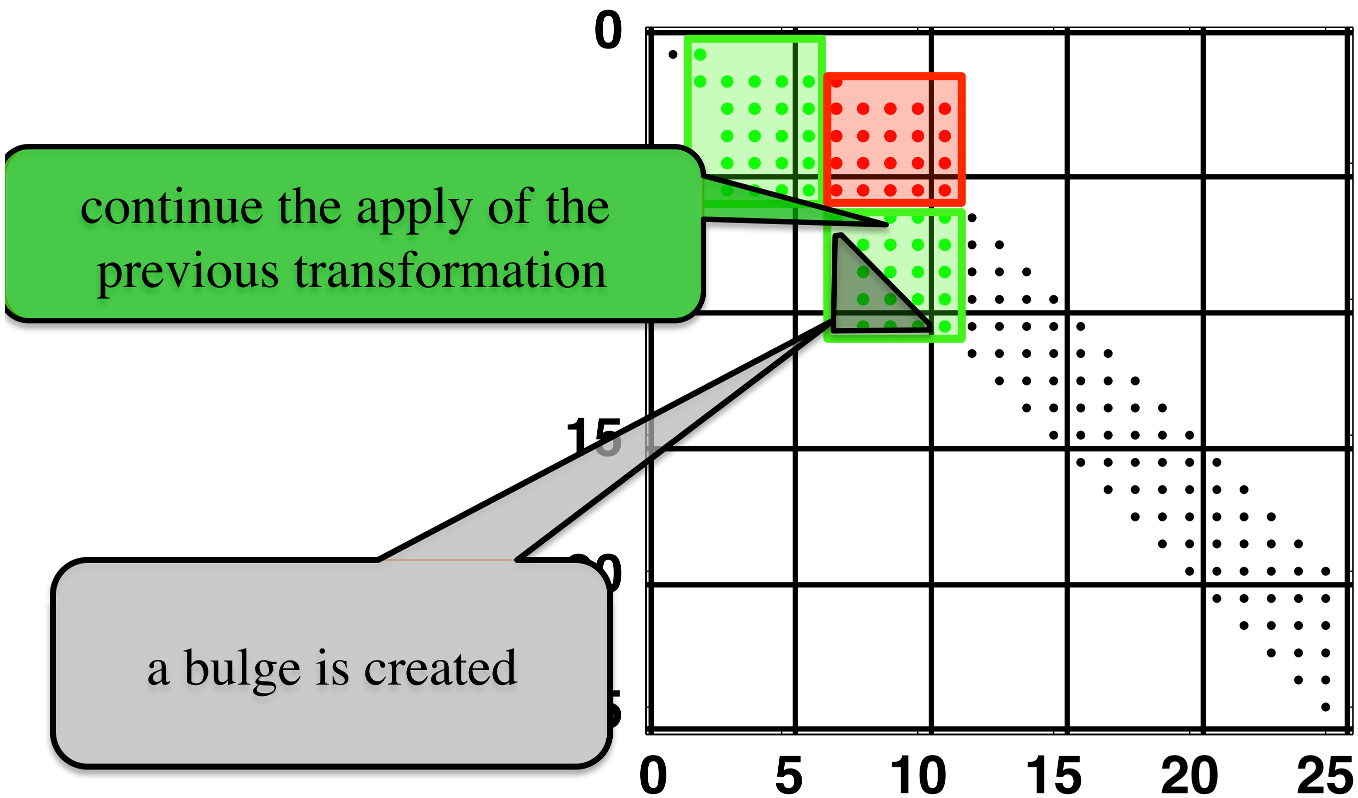


The PLASMA reduction: stage 2

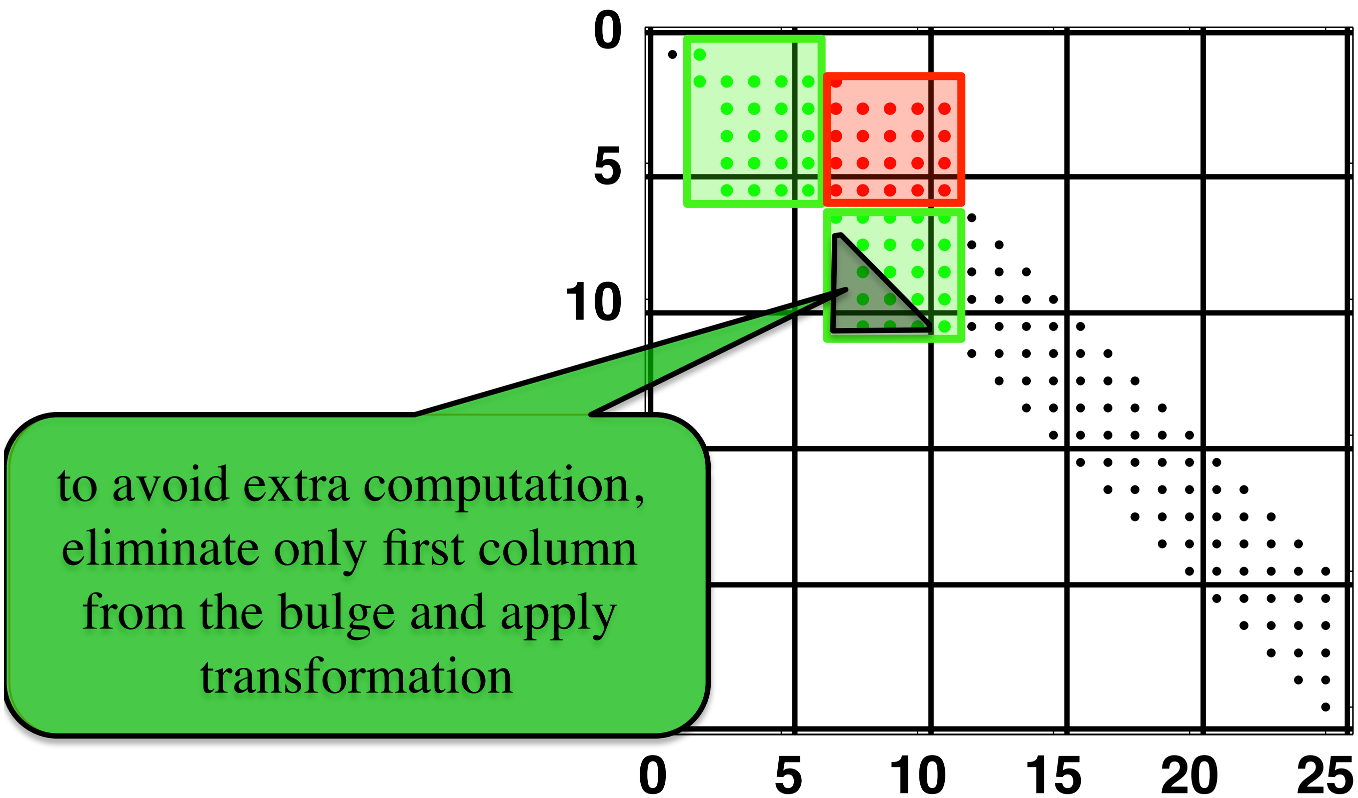


• the red block of data is small (nbxnb), also these operations are unrolled within one kernel

The PLASMA reduction: stage 2

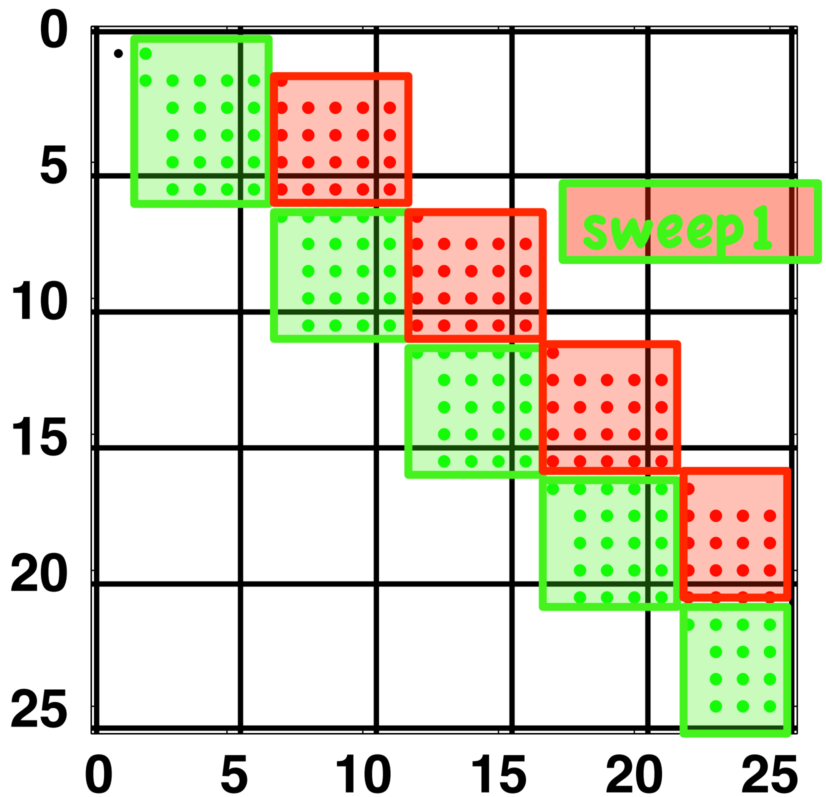


The PLASMA reduction: stage 2



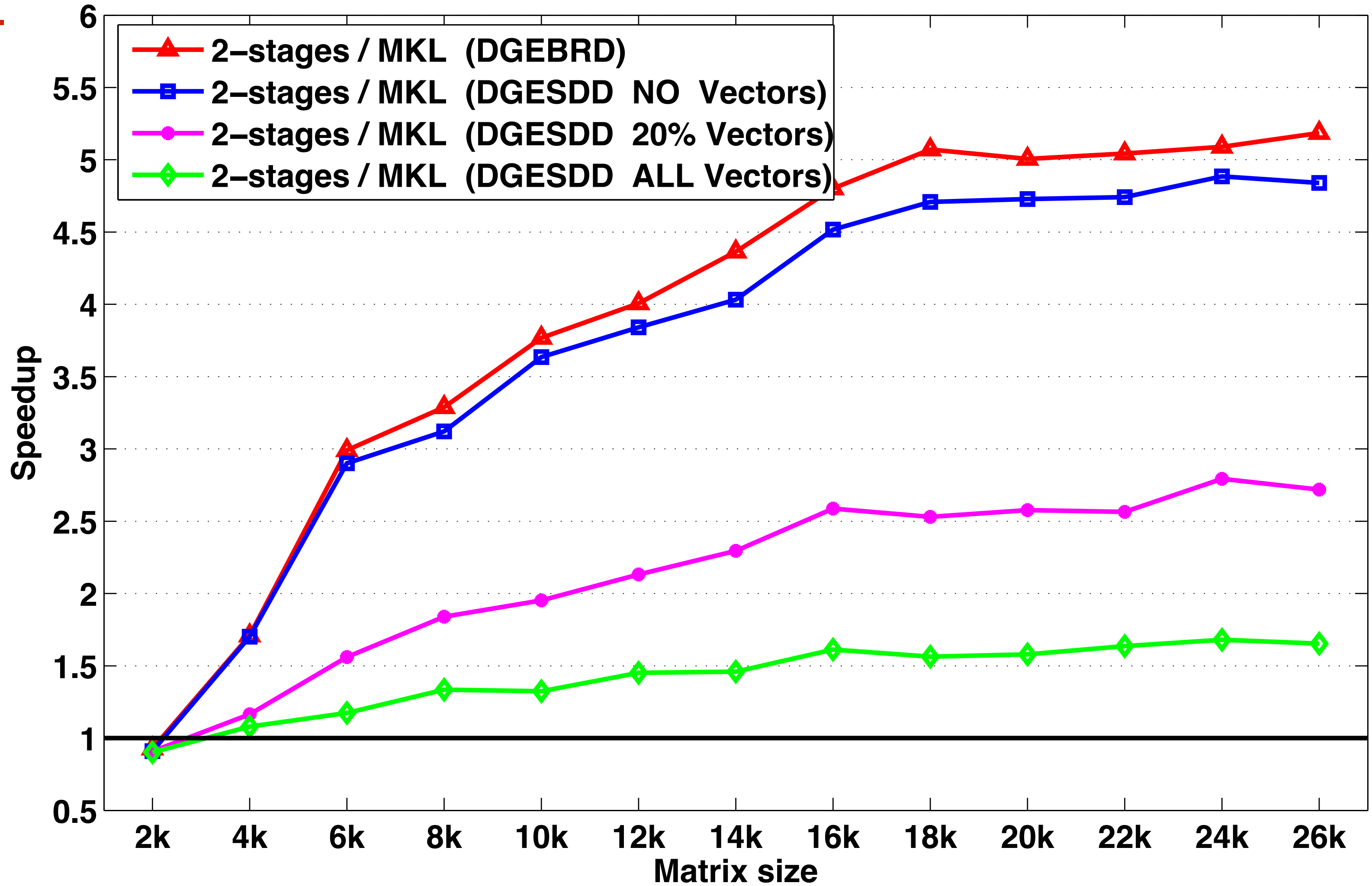
- to increase cache reuse all of these operations are unrolled within one kernel

The PLASMA reduction: stage 2



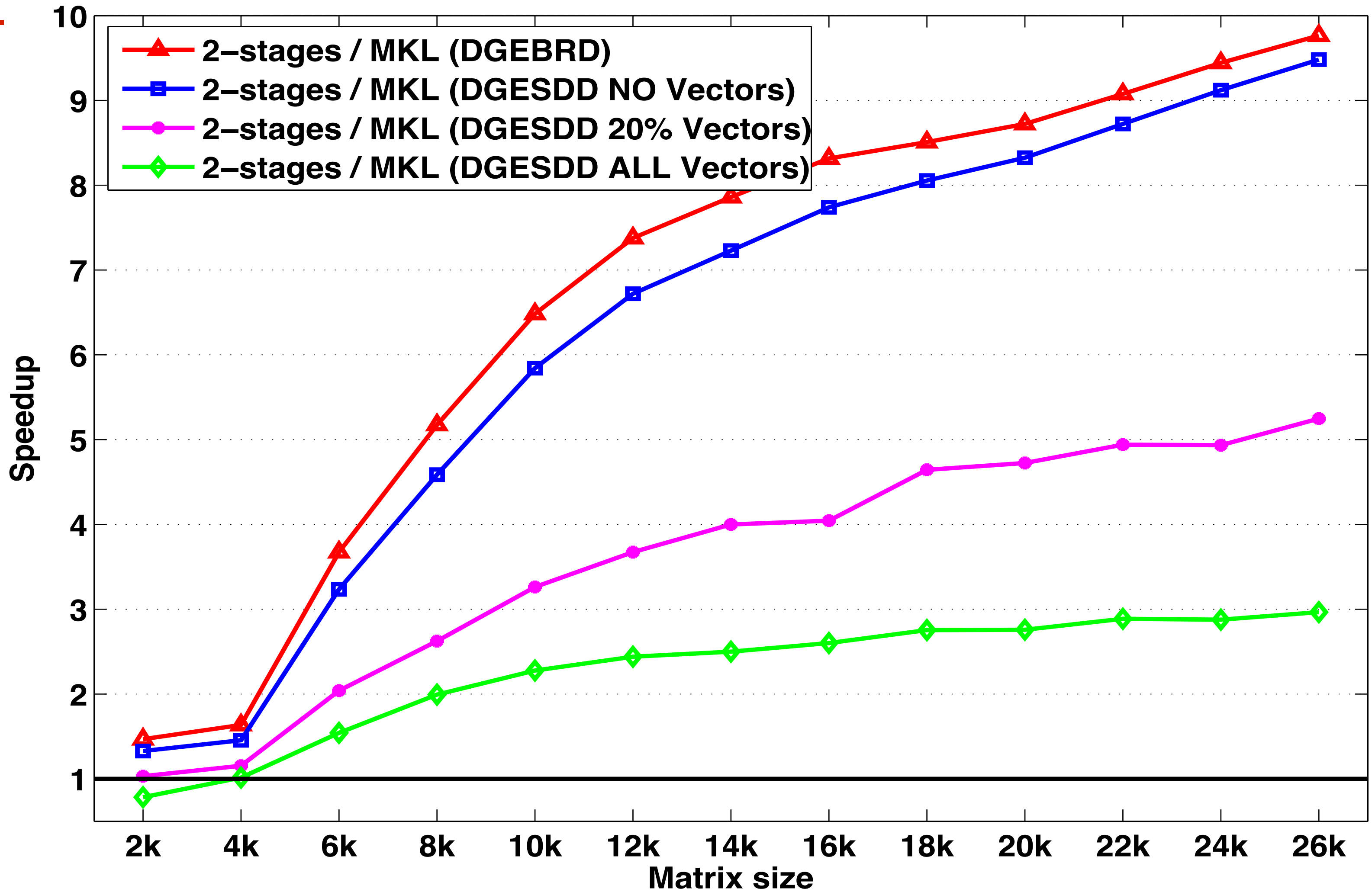
• and so on…. this succession eliminate a sweep

The PLASMA reduction: 2 stage algorithm DGESDD



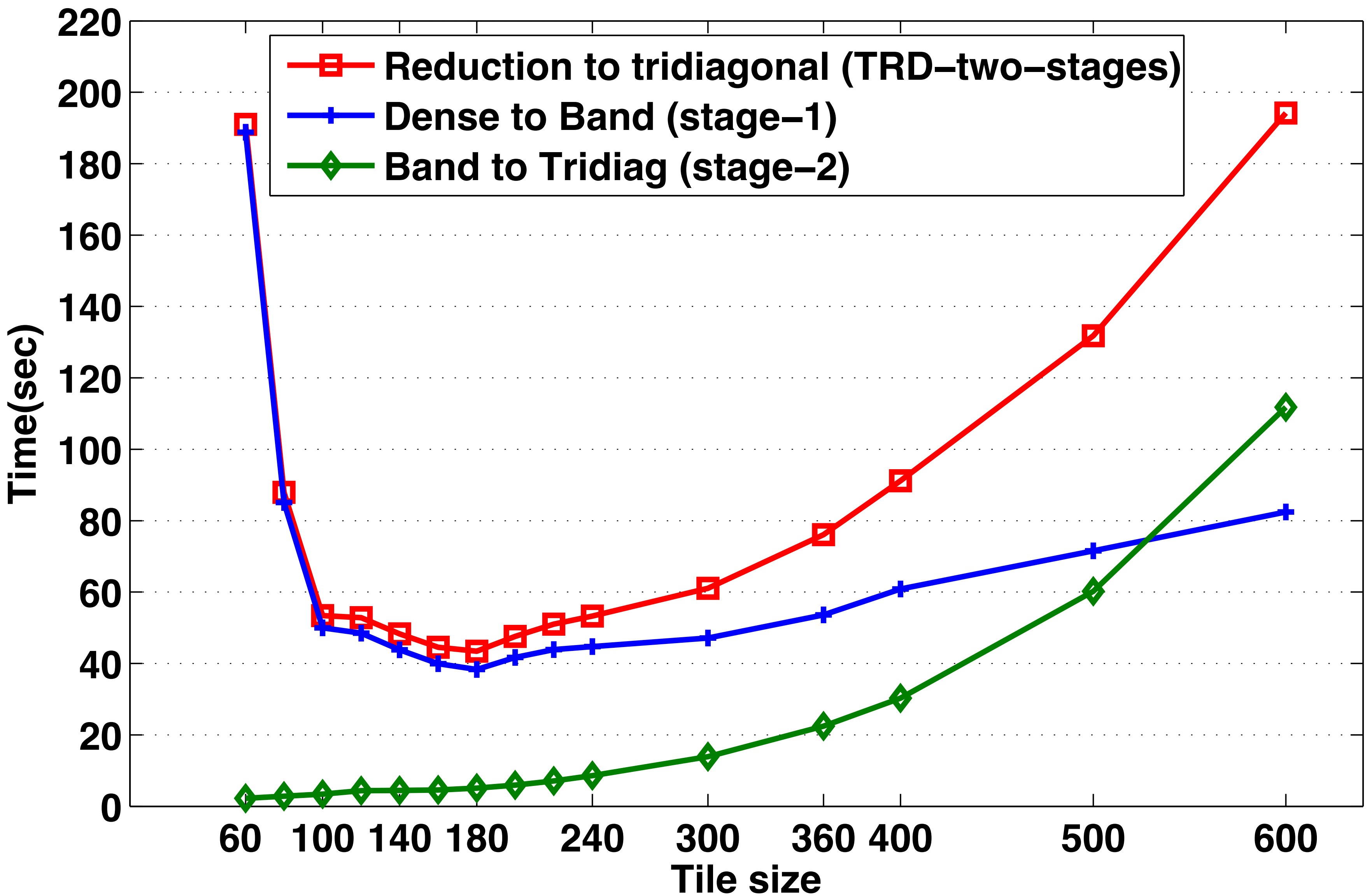
system: 2x8 core Intel Xeon E5-2670 (Sandy Bridge) @ 2.6 GHz

The PLASMA reduction: 2 stage algorithm DGESDD



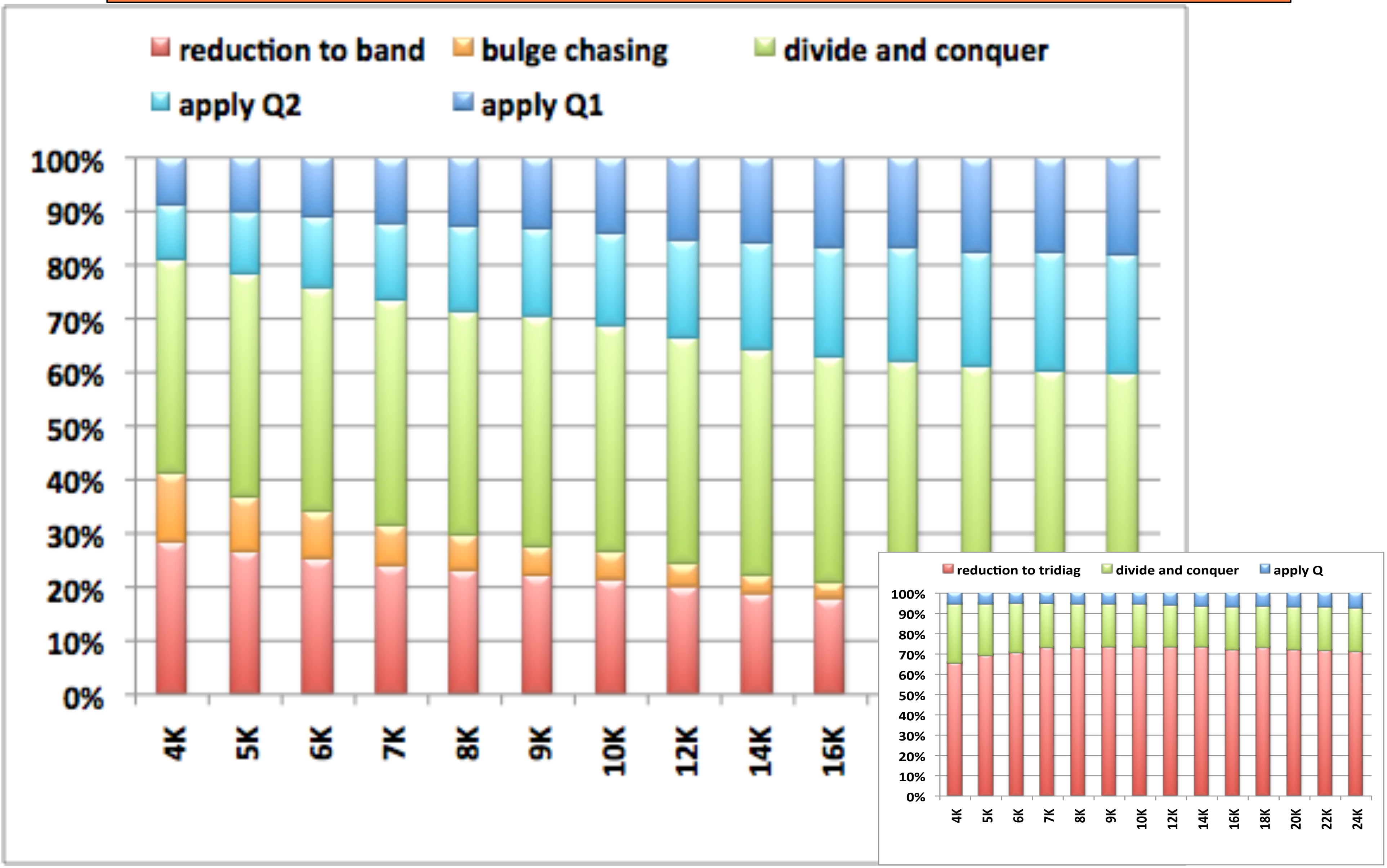
system: 4x12 AMD opteron 6180 SE @ 2.5 GHz

Blocking Matters. What Tile Size?

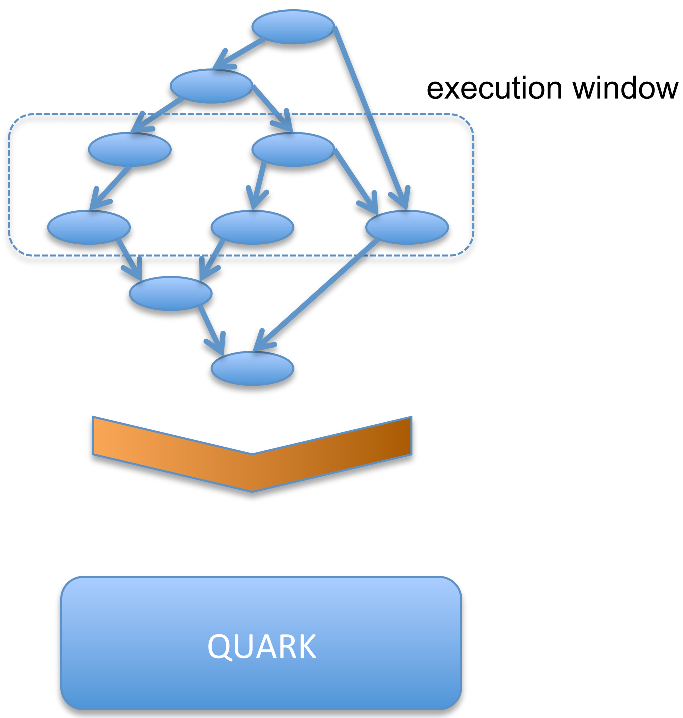


The 2-stage Tridiagonal reduction xSYTRD

The percentage of the time spent in each kernel of the DSYEVDsolver



PLASMA
(On Node)

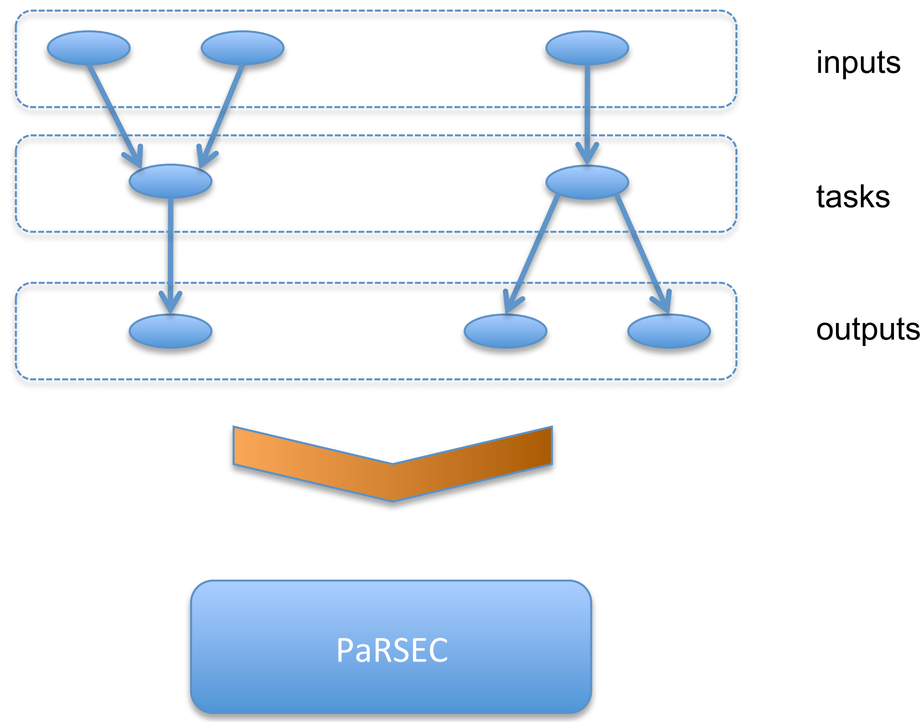


Number of tasks in DAG:

$O(n^3)$

Cholesky: $1/3\ n^3$
LU: $2/3\ n^3$
QR: $4/3\ n^3$

DPLASMA
(Distributed System)



Number of tasks in parameterized DAG:

$O(1)$

Cholesky: 4 (POTRF, SYRK, GEMM, TRSM)
LU: 4 (GETRF, GESSM, TSTRF, SSSSM)
QR: 4 (GEQRT, LARFB, TSQRT, SSRFB)

DAG: Conceptualized & Parameterized

small enough to store on each core in every node = Scalable http://web.eecs.utk.edu/~kurzak/tutorials/

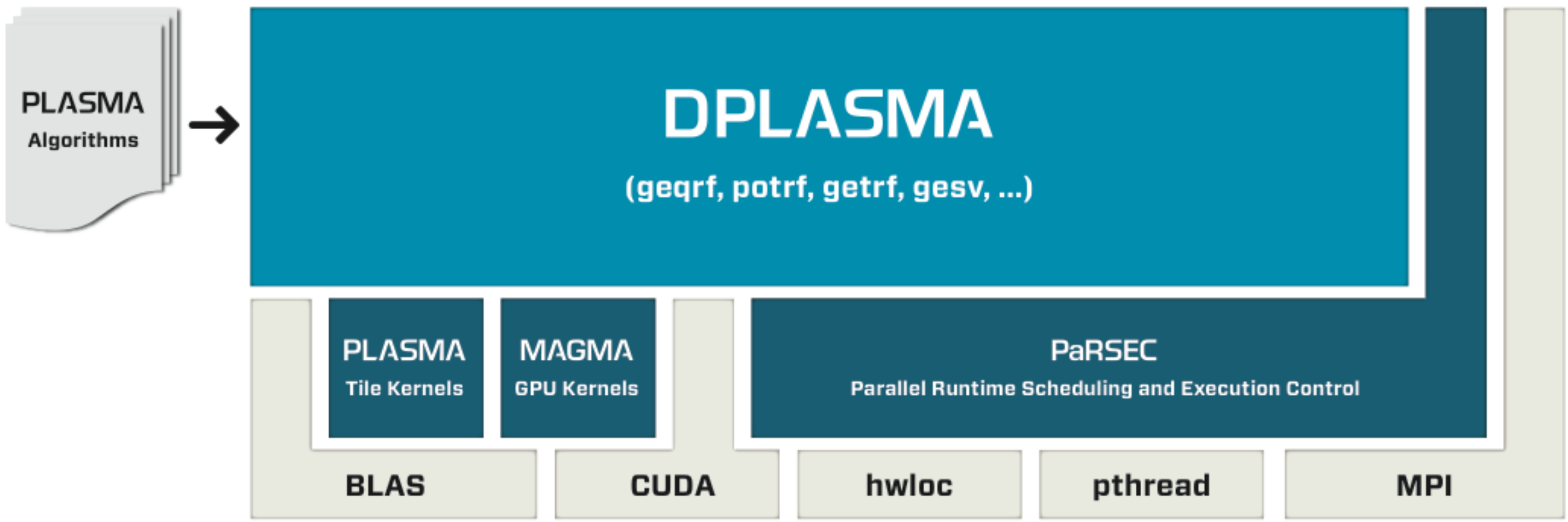
DPLASMA / PaRSEC

Distributed memory PLASMA
/
Parallel Runtime Scheduling and Execution Control

TOC

DPLASMA

Distributed memory PLASMA



A. Bouteiller et al.
Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA
Parallel and Distributed Processing Workshops and Phd Forum - IPDPSW 2011

DPLASMA

Functionality

FUNCTIONALITY	COVERAGE
Linear Systems of Equations	Cholesky, LU (inc. pivoting, PP), LDL (prototype)
Least Squares	QR & LQ
Symmetric Eigenvalue Problem	Reduction to Band (prototype)
Level 3 Tile BLAS	GEMM, TRSM, TRMM, HEMM/SYMM, HERK/SYRK, HER2K/SYR2K

FEATURES

- Covering four precisions: double real, double complex, single real, single complex (D, Z, S, C)
- Providing ScaLAPACK-compatible interface for matrices in F77 column-major layout
- Supporting: Linux, Windows, Mac OS X, UN*X (depends on MPI, hwloc)

PaRSEC

Parallel Runtime Scheduling ane Execution Control

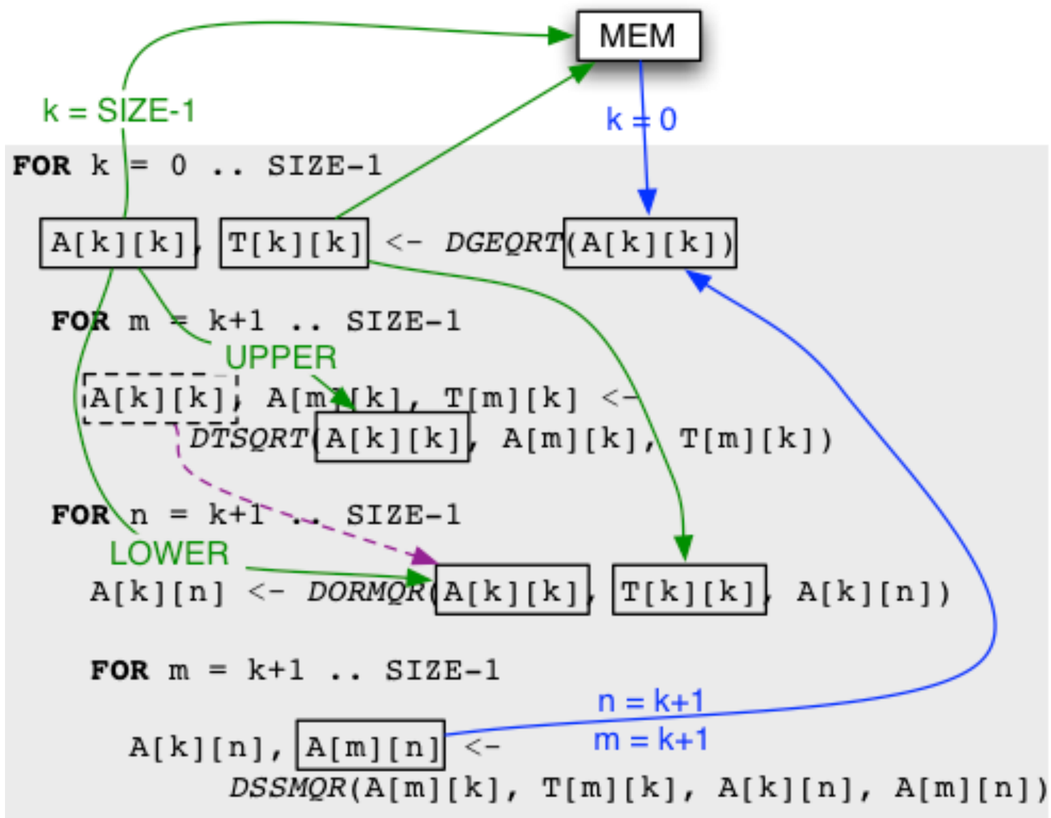
- **Serial definition as the starting poing**

```
FOR k = 0 .. SIZE-1
  A[k][k], T[k][k] <- DGEQRT(A[k][k])
  FOR m = k+1 .. SIZE-1
    A[k][k], A[m][k], T[m][k] <-
         DTSQRT(A[k][k], A[m][k], T[m][k])
  FOR n = k+1 .. SIZE-1
    A[k][n] <- DORMQR(A[k][k], T[k][k], A[k][n])
    FOR m = k+1 .. SIZE-1
      A[k][n], A[m][n] <-
           DSSMQR(A[m][k], T[m][k], A[k][n], A[m][n])
```

PaRSEC

Parallel Runtime Scheduling ane Execution Control

- **Translation to PTG through symbolic analysis**



PaRSEC

Parallel Runtime Scheduling ane Execution Control

FOR k=0 TO N-1
  DGEQRT($_{inout}A_{kk}$)
  FOR n=k+1 to N
    DORMQR($_{in}A$◩$_{kk,\ inout}A_{kn}$)
  FOR m=k+1 to N
    DTSQRT($_{inout}A$◪$_{kk,\ inout}A_{mk}$)
    FOR n=k+1 to N
      DTSMQR($_{in}A_{mk,\ inout}A_{kn,\ inout}A_{mn}$)

serial

PTG

a.k.a Job Dependency Format (JDF)

$DGEQRT_{kkk}$
$1_{ARG} \leftarrow A_{k,k}$ | $DTSMQR_{k,k,k-1}$
$1_{ARG} \Rightarrow DORMQR_{k,k+1..N,k}$(◩)
$1_{ARG} \Rightarrow DTSQRT_{k+1,k,k}$(◪)
$1_{ARG} \Rightarrow A_{k,k}$(◪)

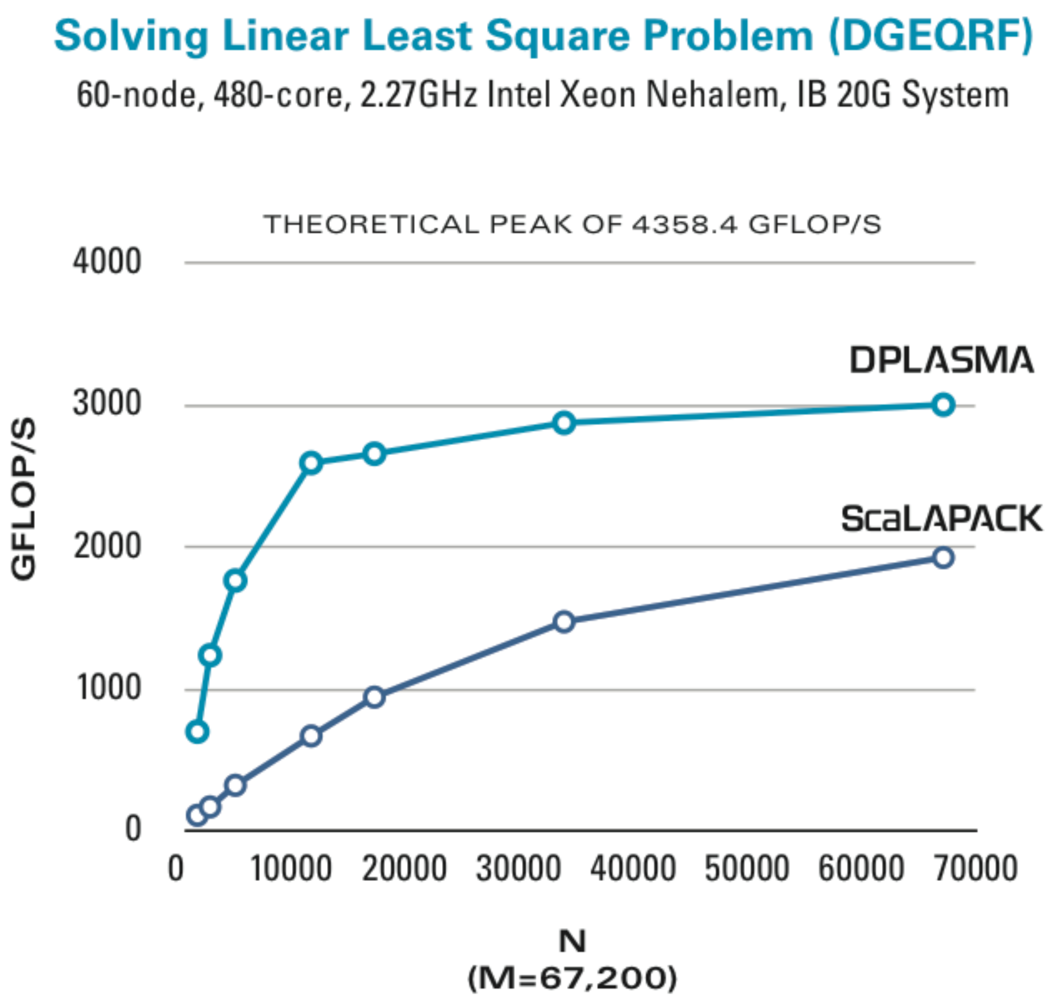
DORMQR$_{knk}$
$1_{ARG} \leftarrow DGEQRT_{k,k,k}$(◩)
$2_{ARG} \leftarrow A_{k,n}$ | $DTSMQR_{k,n,k-1}$
$2_{ARG} \Rightarrow DTSMQR_{k+1,n,k}$
$2_{ARG} \Rightarrow A_{k,n}$

DTSQRT$_{mkk}$
$1_{ARG} \leftarrow DGEQRT_{m-1,k,k}$(◪) | $DTSQRT_{m-1,k,k}$(◪)
$1_{ARG} \Rightarrow DTSQRT_{m+1,k,k}$(◪) | $A_{k,k}$(◪)
$2_{ARG} \leftarrow A_{m,k}$ | $DTSMQR_{m,k,k-1}$
$2_{ARG} \Rightarrow DTSMQR_{m,k+1..N,k}$
$2_{ARG} \Rightarrow A_{m,k}$

DTSMQR$_{mnk}$
$1_{ARG} \leftarrow DTSQRT_{m,k,k}$
$2_{ARG} \leftarrow DORMQR_{m-1,n,k}$ | $DTSMQR_{m-1,n,k}$
$2_{ARG} \Rightarrow DTSMQR_{m+1,n,k}$ | $A_{n,k}$
$3_{ARG} \leftarrow A_{m,n}$ | $DTSMQR_{m,n,k-1}$
$3_{ARG} \Rightarrow DGEQRT_{m,n,k+1}$ | *DORMQR*$_{m,n,k+1}$ |
$\Rightarrow DTSQRT_{m,n,k+1}$ | *DTSMQR*$_{m,n,k+1}$ |
$\Rightarrow A_{m,n}$

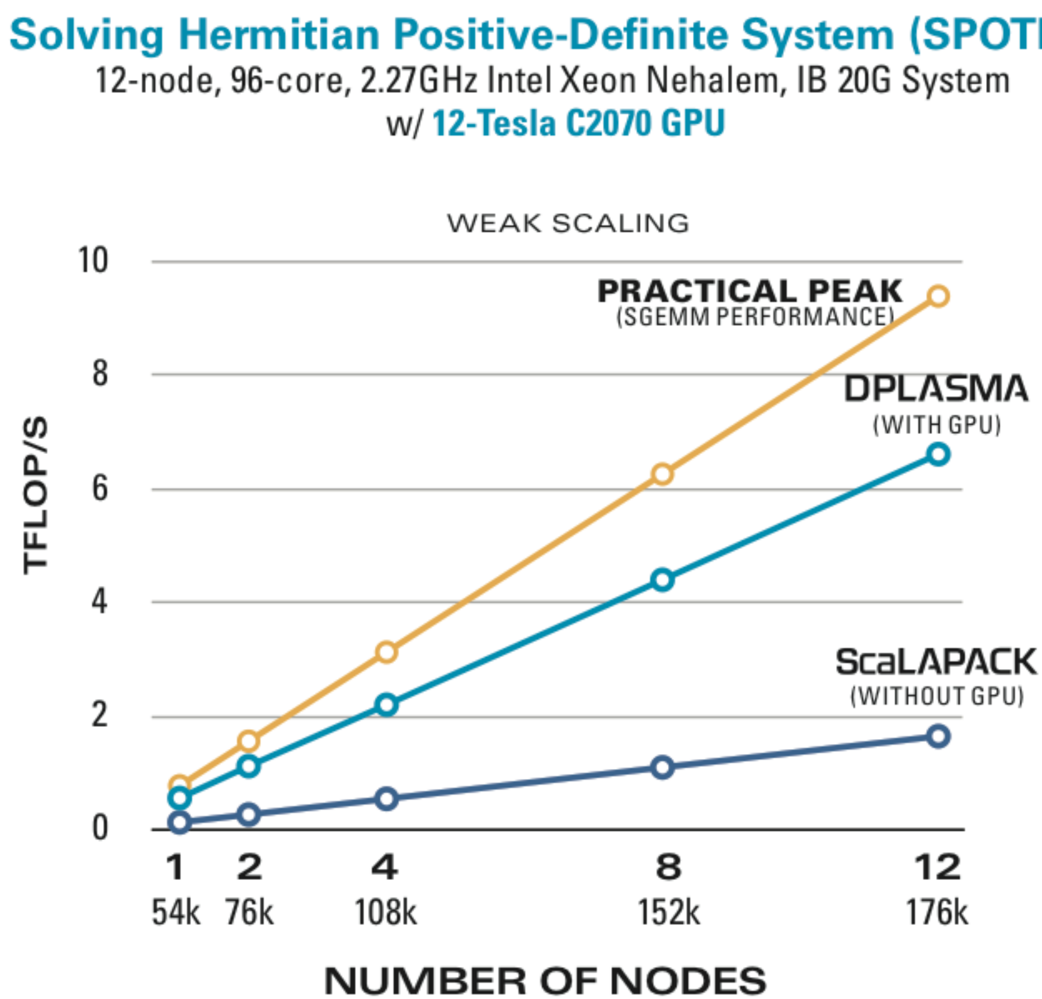
DPLASMA / PaRSEC

performance



DPLASMA / PaRSEC

performance



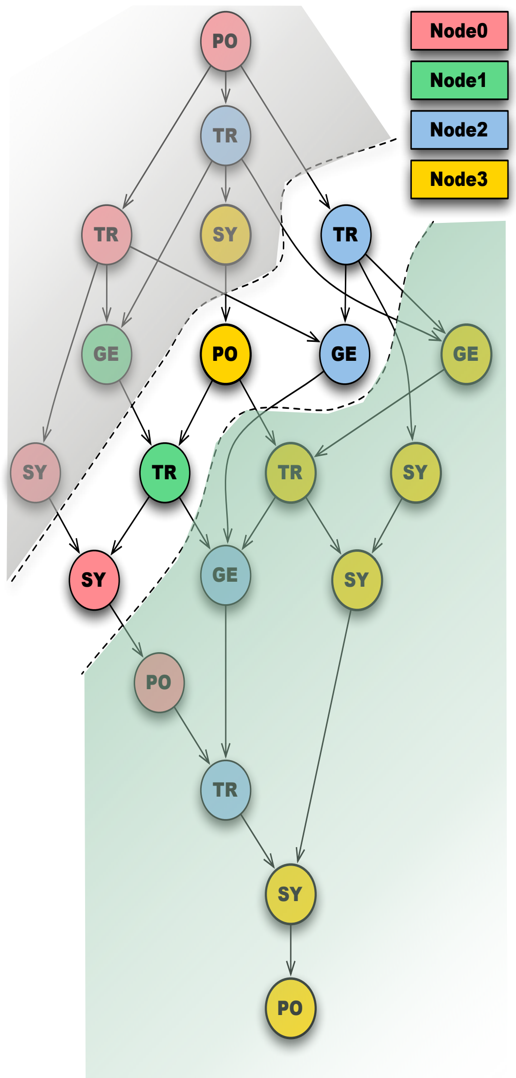
MATRIX SIZE (NxN)

Distributed Memory Runtime System

- **Pa**r**a**llel **R**untime **S**cheduler & **E**xecution **C**ontrol
  - Executes a **dataflow** representation of a program
  - **Scheduler** provides
    - Automatic **load-balance between cores**
    - Harness the power of accelerators (GPU, Mic, etc)
  - Works on large scale distributed memory machines
    - **Communications** are **implicit**, **overlapped**
    - user defined Communication pattern and data-distribution

Prominent feature: *Parameterized Task Graph*

Runtime DAG scheduling



- Every node has the symbolic DAG representation
  - Only the (node local) frontier of the DAG is considered
  - Distributed Scheduling based on remote completion notifications
- Background remote data transfer automatic with overlap
- NUMA / Cache aware Scheduling
  - Work Stealing and sharing based on memory hierarchies

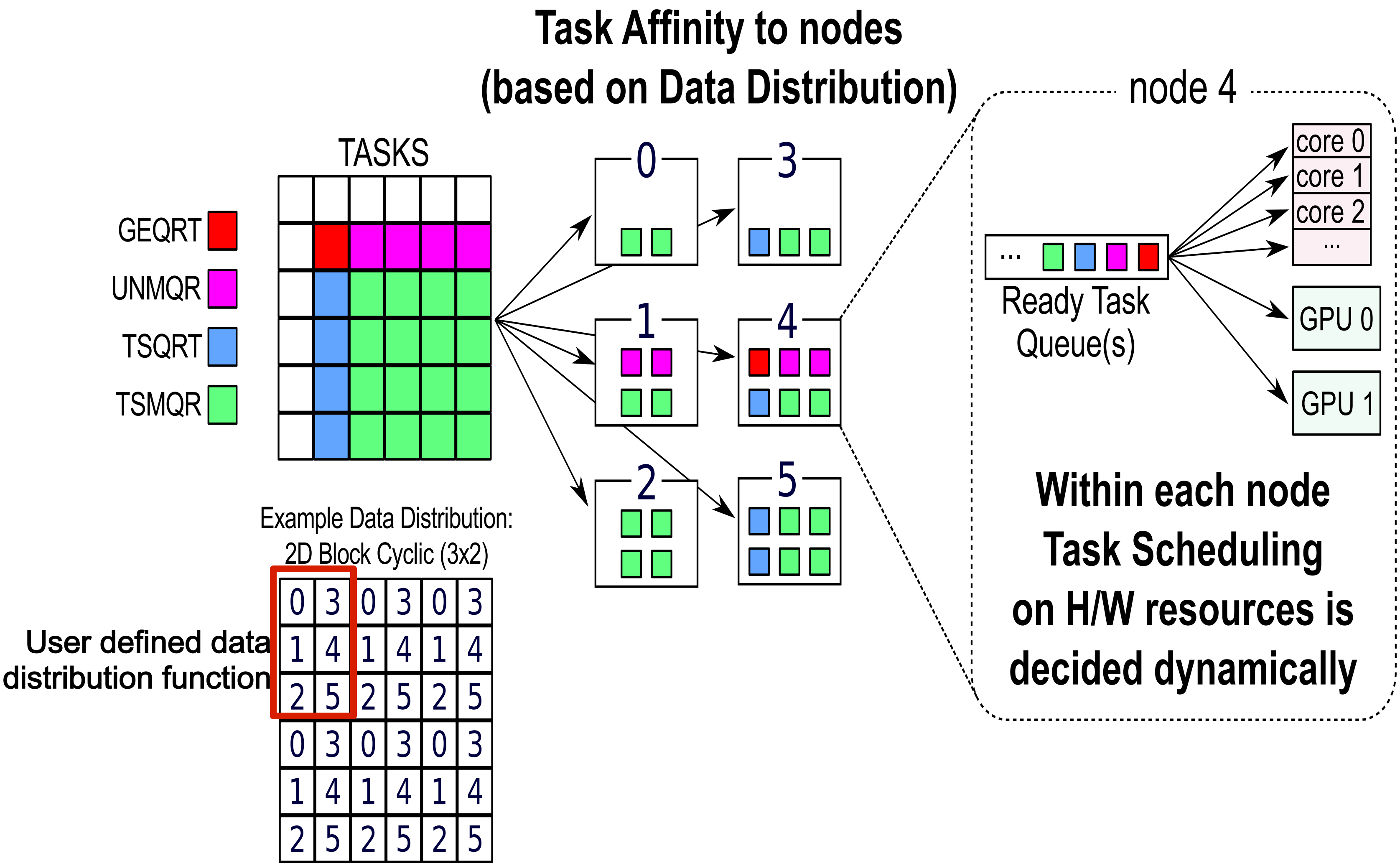
Related Work

	PaRSEC	SMPss	StarPU	Charm ++	FLAME	QUARK	Tblas	PTG
Scheduling	Distr. (1/core)	Repl (1/node)	Repl (1/node)	Distr. (Actors)	w/ SuperMatrix	Repl (1/node)	Centr.	Centr.
Language	Internal or Seq. w/ Affine Loops or w/ add_task	Seq. w/ add_tas k	Seq. w/ add_task	Msg-Driven Objects	Internal (LA DSL)	Seq. w/ add_task	Seq. w/ add_task	Internal
Accelerator	GPU	GPU	GPU		GPU	GPU		
Availability	Public	Public	Public	Public	Public	Public	Not Avail.	Not Avail.

Early stage: ParalleX
Non-academic: Swarm, MadLINQ, CnC

All projects support Distributed and Shared Memory
(QUARK with QUARKd; FLAME with Elemental)

Task Affinity in PaRSEC



International Community Effort

- We believe this needs to be an international collaboration for various reasons including:
  - The scale of investment
  - The need for international input on requirements
  - US, Europeans, Asians, and others are working on their own software that should be part of a larger vision for HPC.
  - No global evaluation of key missing components
  - Hardware features are uncoordinated with software development

Summary

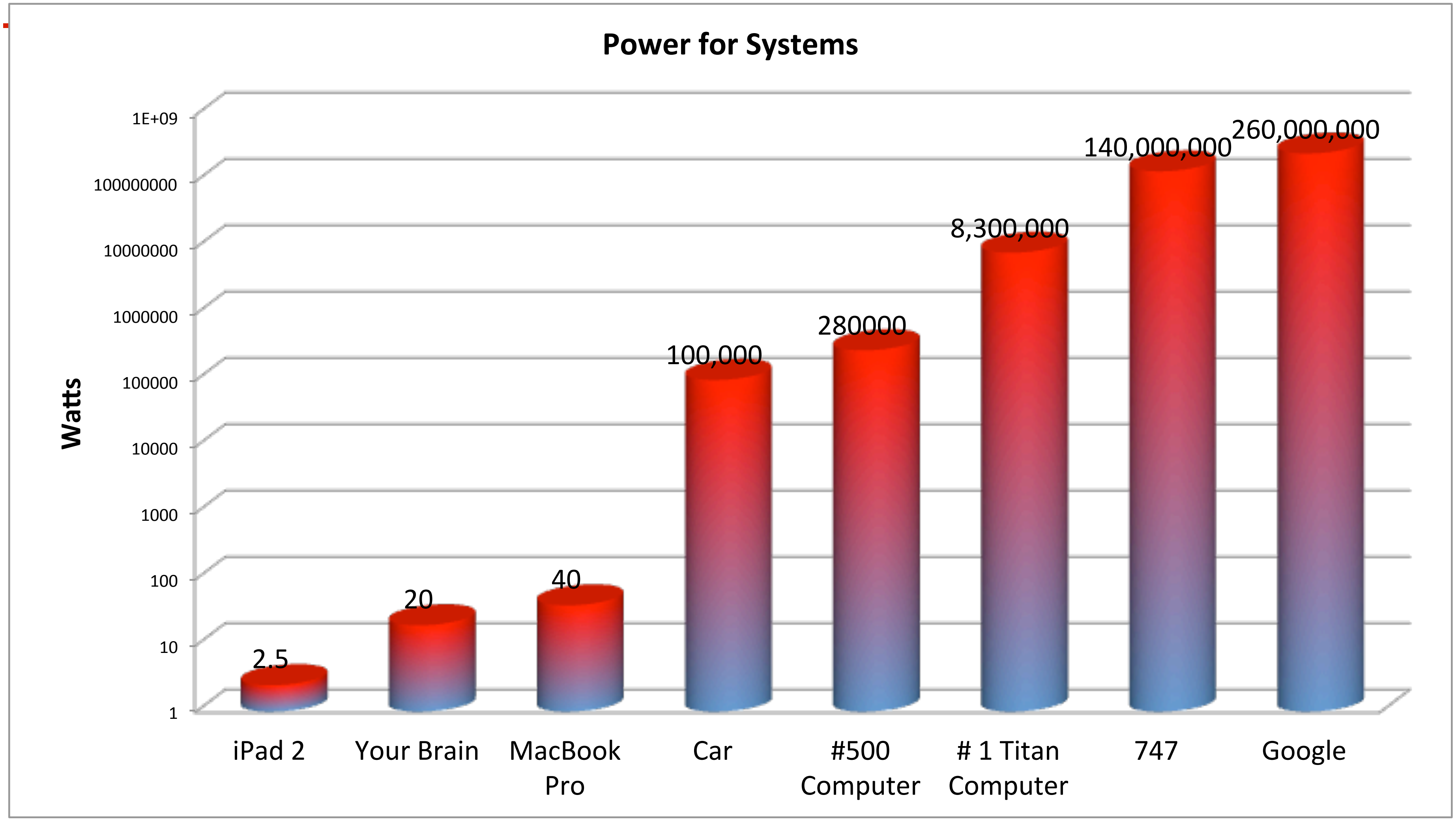
- Major Challenges are ahead for extreme computing
  - Parallelism $O(10^9)$
    - Programming issues
  - Hybrid
    - Peak and HPL may be very misleading
    - No where near close to peak for most apps
  - Fault Tolerance
    - Today Sequoia BG/Q node failure rate is 1.25 failures/day
  - Power
    - 50 Gflops/w (today at 2 Gflops/w)
- We will need completely new approaches and technologies to reach the Exascale level

Collaborators / Software / Support

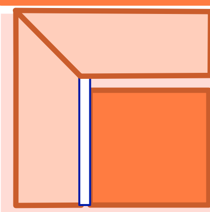
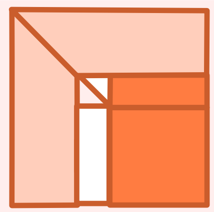
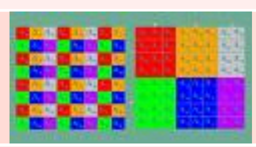
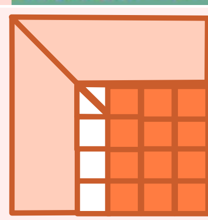
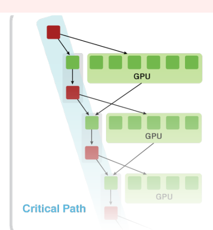
- **PLASMA**
  http://icl.cs.utk.edu/plasma/
- **MAGMA**
  http://icl.cs.utk.edu/magma/
- **Quark (RT for Shared Memory)**
  - **http://icl.cs.utk.edu/quark/**
- **PaRSEC(**Parallel Runtime Scheduling and Execution Control**)**
  - **http://icl.cs.utk.edu/parsec/**

- Collaborating partners
  University of Tennessee, Knoxville
  University of California, Berkeley
  University of Colorado, Denver

  INRIA, France
  KAUST, Saudi Arabia

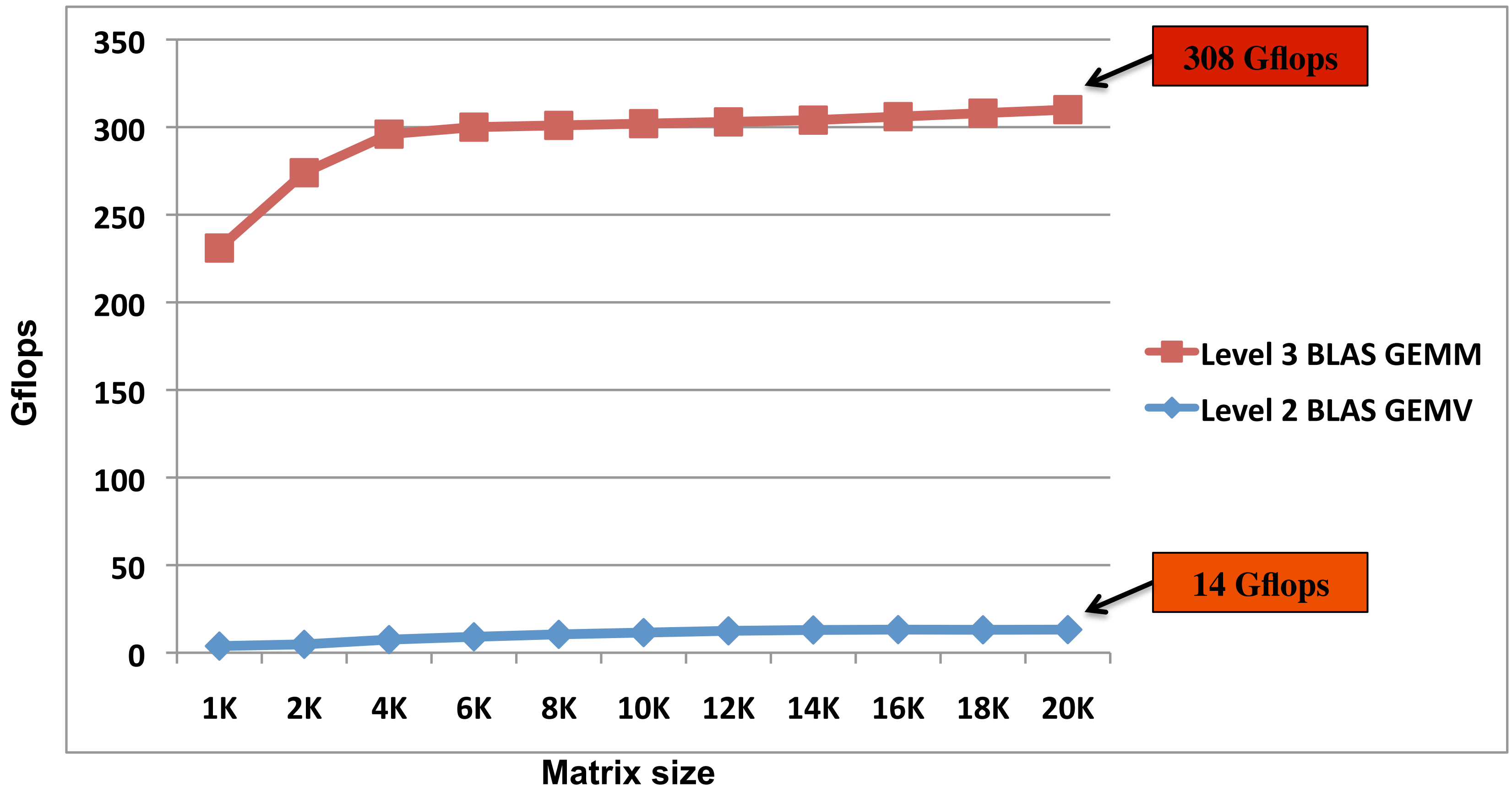

Power for Systems
Watts
1E+09
100000000
10000000
1000000
100000
10000
1000
100
10
1
2.5
20
40
100,000
280000
8,300,000
140,000,000
260,000,000
iPad 2
Your Brain
MacBook Pro
Car
#500 Computer
1 Titan Computer
747
Google

A New Generation of DLA Software

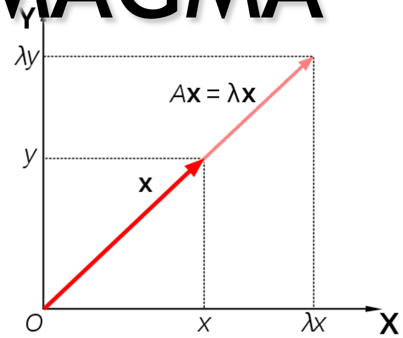
Software/Algorithms follow hardware evolution in time		
LINPACK (70's) (Vector operations)		Rely on - Level-1 BLAS operations
LAPACK (80's) (Blocking, cache friendly)		Rely on - Level-3 BLAS operations
ScaLAPACK (90's) (Distributed Memory)		Rely on - PBLAS Mess Passing
PLASMA New Algorithms (many-core friendly)		Rely on - a DAG/scheduler - block data layout - some extra kernels
MAGMA Hybrid Algorithms (heterogeneity friendly)		

Performance of Level 2 and Level 3 BLAS

- 2 – 8 cores Intel Xeon E5-2670 (Sandy Bridge), 2.6 GHz.
  24 MB shared L3 cache, and each core has a private 256 KB L2 and 64 KB L1.
  Theoretical peak for this architecture in double precision is 20.8 Gflop/s per core (333 Gflops total).
  8 flop/cycle*2.6 cycle/sec*16 cores = 332.8 Gflop/s
  Compiled with gcc 4.4.6 and using MKL_composer_xe_2013.3.163

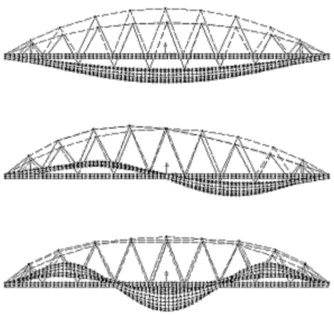


Eigenproblem Solvers in MAGMA



- $A\,x = \lambda\,x$
  - Quantum mechanics (Schrödinger equation)
  - Quantum chemistry
  - Principal component analysis (in data mining)
  - Vibration analysis (of mechanical structures)
  - Image processing, compression, face recognit
  - Eigenvalues of graph, e.g., in Google's page r

    . . .



- **Need to solve it fast**

Current MAGMA results:
MAGMA with 1 GPU can be 12x faster vs vendor libraries on state-of-art multicore systems

T. Dong, J. Dongarra, S. Tomov, I. Yamazaki, T. Schulthess, and R. Solca, *Symmetric dense matrix-vector multiplication on multiple GPUs and its application to symmetric dense and sparse eigenvalue problems*, ICL Technical report, 03/2012.

J. Dongarra, A. Haidar, T. Schulthess, R. Solca, and S. Tomov, *A novel hybrid CPU- GPU generalized eigensolver for electronic structure calculations based on fine grained memory aware tasks*, ICL Technical report, 03/2012.

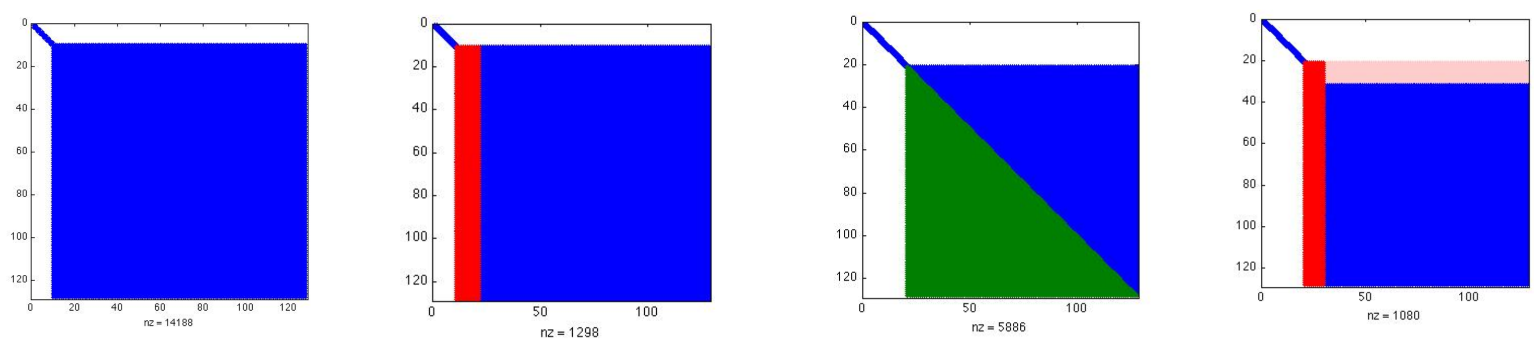
The Standard Tridiagonal Reduction xSYTRD

- **LAPACK xSYTRD:**

1. Apply left-right transformations Q A Q* to the panel $\begin{pmatrix} A_{22} \\ A_{32} \end{pmatrix}$
2. Update the remaining submatrix A_{33}

$$\begin{pmatrix} T_{11} & T_{21}^T & 0 \\ T_{21} & A_{22} & A_{32}^T \\ 0 & A_{32} & A_{33} \end{pmatrix} \equiv \begin{pmatrix} T_{11} & T_{21}^T & 0 \\ T_{21} & A_{22} & A_{32}^T \\ 0 & A_{32} & A_{33} \end{pmatrix} \Longrightarrow \begin{pmatrix} T_{11} & T_{21}^T & 0 \\ T_{21} & T_{22} & T_{23}^T \\ 0 & T_{23} & A_{33} \end{pmatrix}$$

where $A_{33} = A_{33} - YW^T - WY^T$



step k: Q A Q* then update → step k+1

For the symmetric eigenvalue problem:
First stage takes:
- 90% of the time if only eigenvalues
- 50% of the time if eigenvalues and eigenvectors

The Standard Tridiagonal Reduction xSYTRD

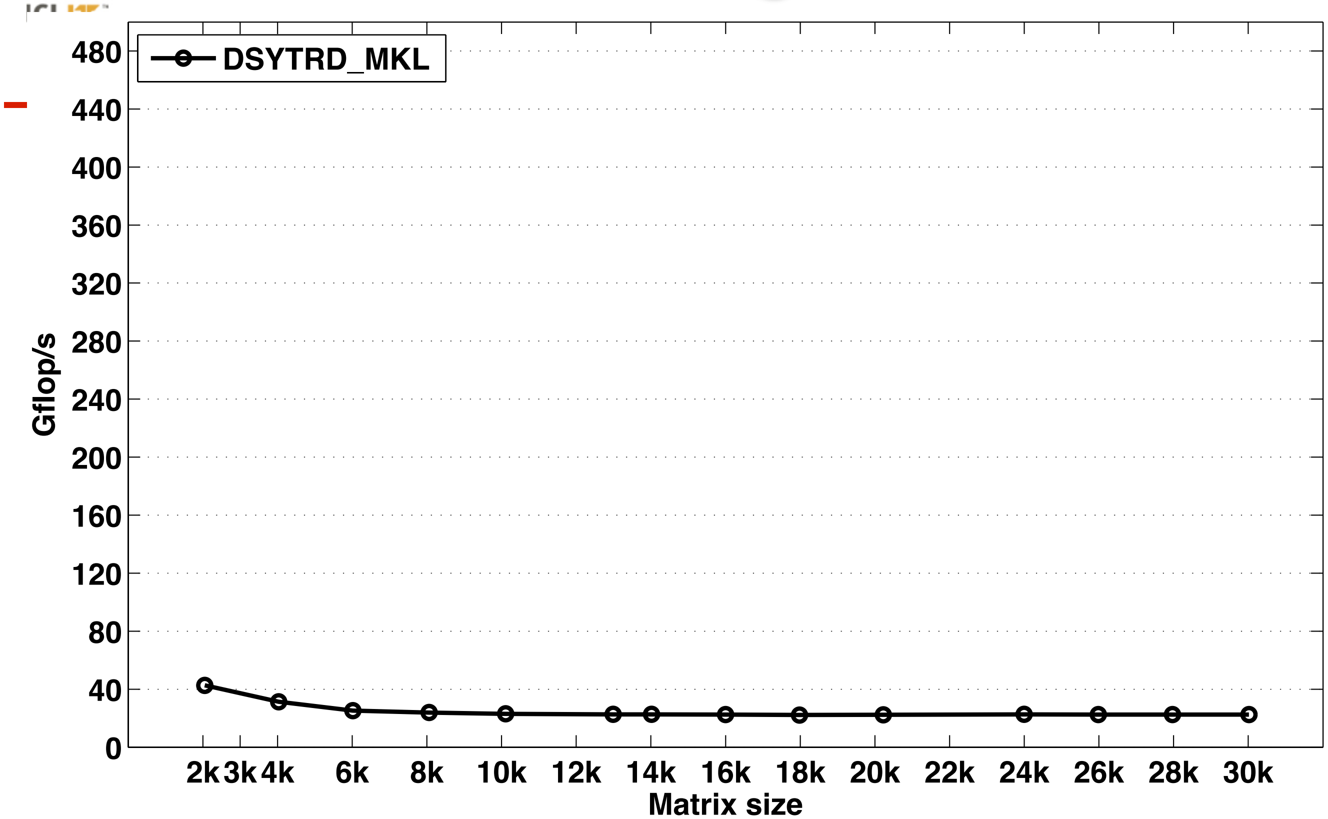
✷ Characteristics

1. Phase 1 requires :
   - 4 panel vector multiplications,
   - 1 symmetric matrix vector multiplication with A_{33},
   - Cost $2(n-k)^2b$ Flops.
2. Phase 2 requires:
   - Symmetric update of A_{33} using SYRK,
   - Cost $2(n-k)^2b$ Flops.

✷ Observations

- Too many Level 2 BLAS ops,
- Relies on panel factorization,
- Total cost $4n^3/3$
- ➔Bulk sync phases,
- ➔Memory bound algorithm.

Toward fast Eigensolver



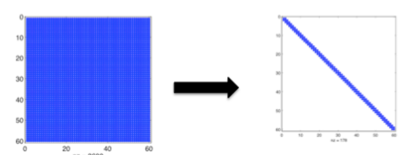
flops formula: $n^3/3$*time

Higher is faster

Keeneland system, using one node
3 NVIDIA GPUs (M2090@ 1.1 GHz, 5.4 GB)
2 x 6 Intel Cores (X5660 @ 2.8 GHz, 23 GB)

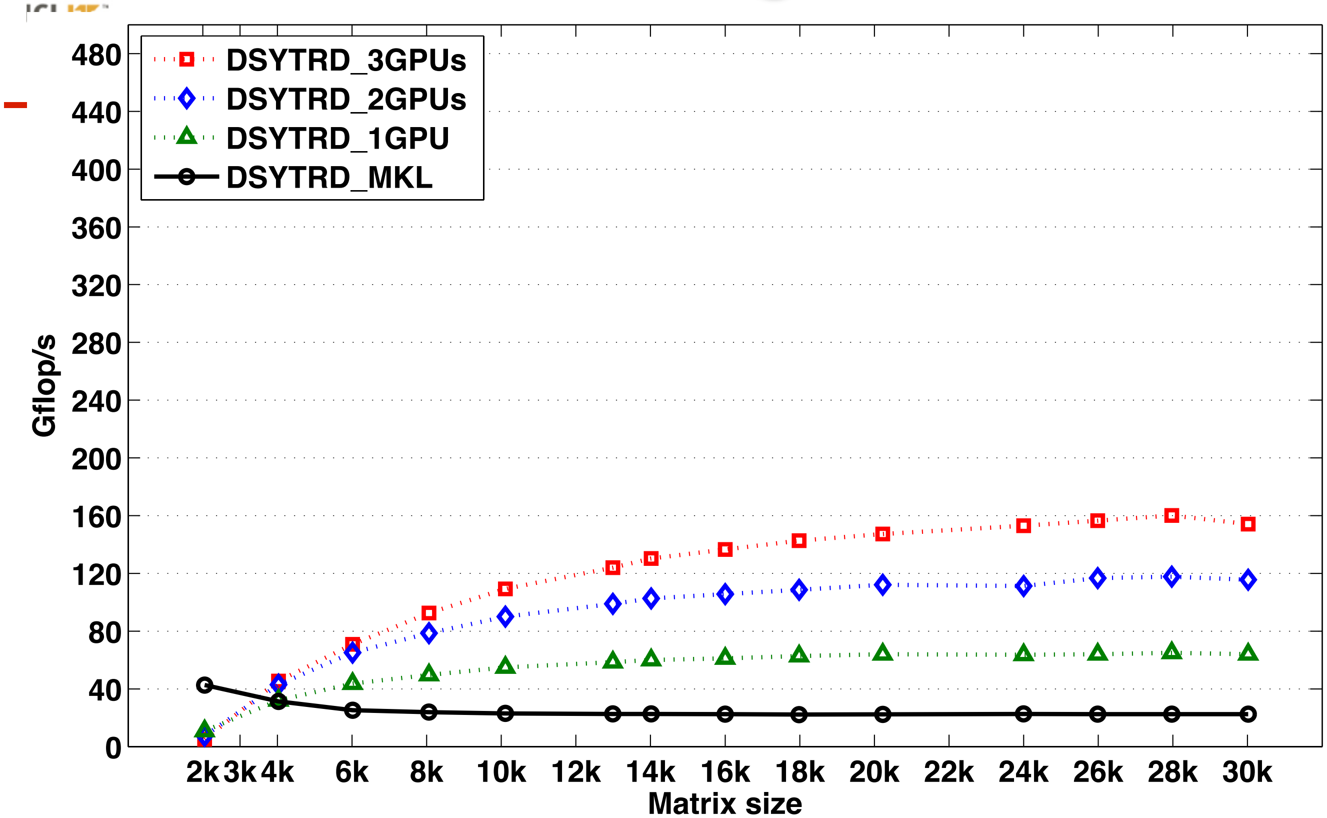
✸ Characteristics

- Too many Blas-2 op,
- Relies on panel factorization,
- ➔Bulk sync phases,
- ➔Memory bound algorithm.



A. Haidar, S. Tomov, J. Dongarra, T. Schulthess, and R. Solca, *A novel hybrid CPU-GPU generalized eigensolver for electronic structure calculations based on fine grained memory aware tasks*, ICL Technical report, 03/2012.

Toward fast Eigensolver



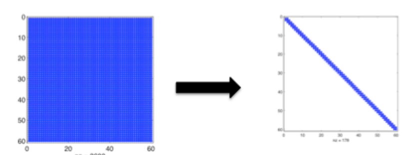
flops formula: $n^3/3$*time

Higher is faster

Keeneland system, using one node
3 NVIDIA GPUs (M2090@ 1.1 GHz, 5.4 GB)
2 x 6 Intel Cores (X5660 @ 2.8 GHz, 23 GB)

Characteristics

- Blas-2 GEMV moved to the GPU,
- Accelerate the algorithm by doing all BLAS-3 on GPU,
- ➔Bulk sync phases,
- ➔Memory bound algorithm.



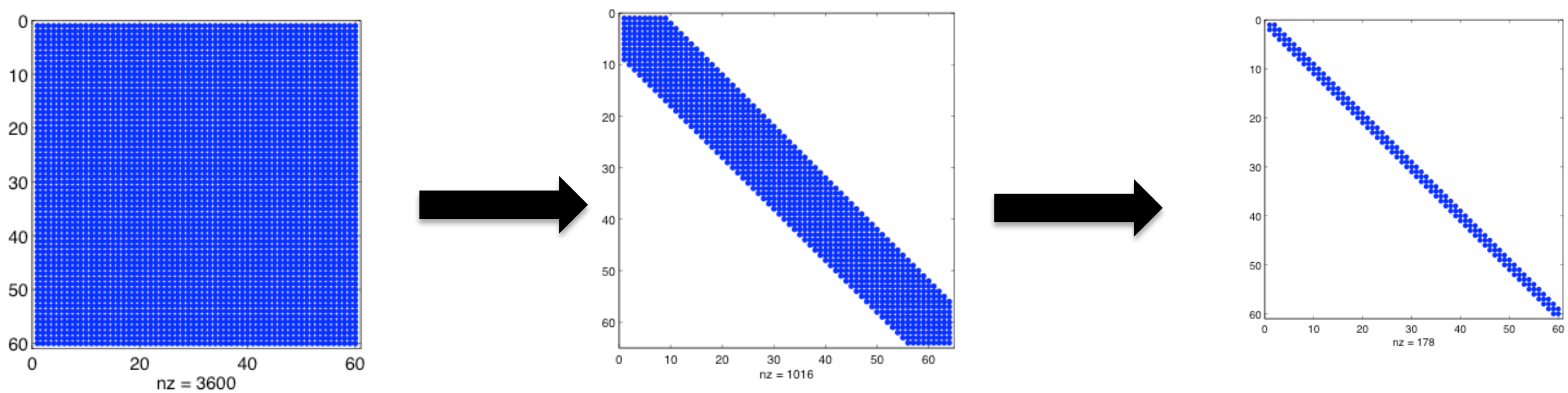
A. Haidar, S. Tomov, J. Dongarra, T. Schulthess, and R. Solca, *A novel hybrid CPU-GPU generalized eigensolver for electronic structure calculations based on fine grained memory aware tasks*, ICL Technical report, 03/2012.

Symmetric Eigenvalue Problem

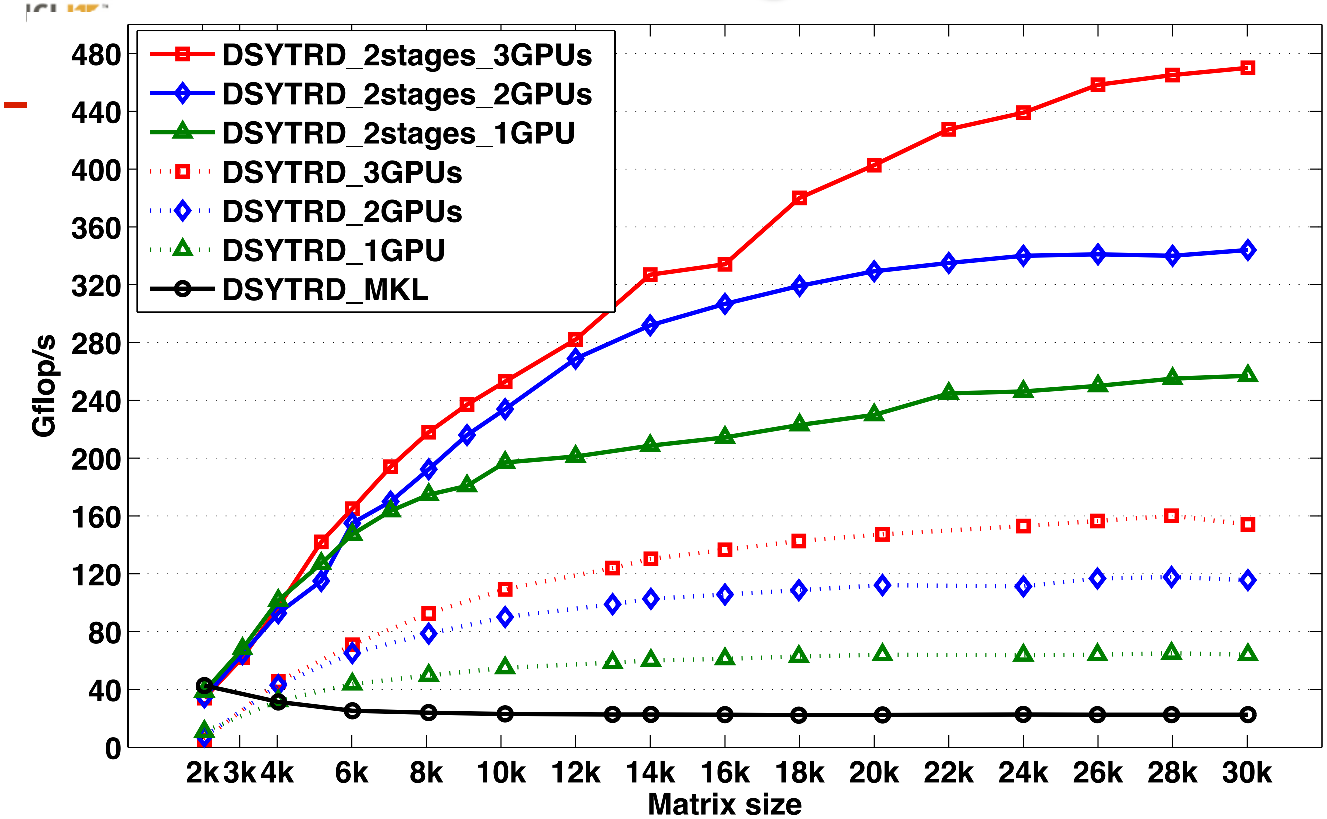
- Standard reduction algorithm is very slow on multicore.

Better Formulation:

- Step1: Reduce the dense matrix to band.
  - Matrix-matrix operations, high degree of parallelism
- Step2: Bulge Chasing on the band matrix
  - by group and cache aware



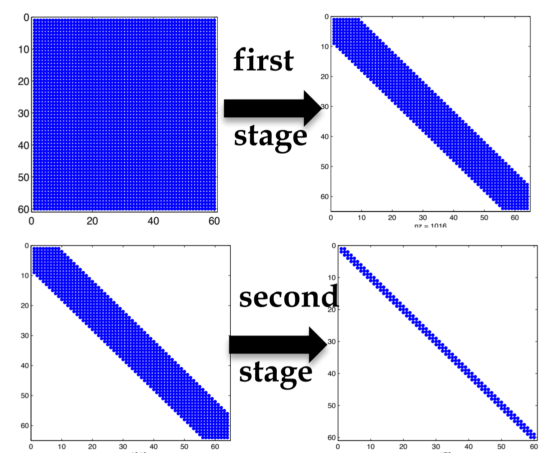
Toward fast Eigensolver



flops formula: $n^3/3$*time

Higher is faster

Keeneland system, using one node
3 NVIDIA GPUs (M2090@ 1.1 GHz, 5.4 GB)
2 x 6 Intel Cores (X5660 @ 2.8 GHz, 23 GB)



✱ Characteristics

- Stage 1: BLAS-3, increasing computational intensity,
- Stage 2: BLAS-1.5, new cache friendly kernel,
- 4X/12X faster than standard approach,
- Bottelneck: if all Eigenvectors are required, it has 1 back transformation extra cost.

A. Haidar, S. Tomov, J. Dongarra, T. Schulthess, and R. Solca, *A novel hybrid CPU-GPU generalized eigensolver for electronic structure calculations based on fine grained memory aware tasks*, ICL Technical report, 03/2012.